Indexing RDF Data using Materialized SPARQL Queries

SPARQL Query Processing and Index Selection

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

Dr. Rer. Nat.
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
M.Sc. Roger Humberto Castillo Espinola

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Dr. Ulf Leser

2. Prof. Johann-Christoph Freytag, Ph.D.

3. Prof. Dr. Adrian Paschke

eingereicht am: 23.11.2011
Tag der mündlichen Prüfung: 04.05.2012

Abstract

The vision of the *Semantic Web* to enrich the hyperlinked information published on the Web has gained a lot of attention of the scientific community. The main idea is to transform the up to now only human readable web pages into computer-processable data by adding semantic metadata that describe resources and relations among them. The *Resource Description Framework (RDF)* has been introduced by the *W3C* as a data model for annotations. RDF encodes information in triples consisting of subject, predicate, and object. A triple specifically describes one property of the subject, given by the object, related to it by means of the predicate. A multitude of triples together builds an RDF dataset. Currently, large RDF datasets are available on the Web that consists of millions of triples. Efficiently querying this information has become a crucial task.

SPARQL is a declarative query language proposed by the W3C to extract information from RDF data. SPARQL queries are formulated in a similar fashion as select-project-join operations in relational databases. However, querying RDF datasets is complex due to the graph structure of the data and the often complex structure of a given query.

In this thesis, we propose to use materialized queries as a special index structure for *RDF* data. We strive to reduce the query processing time by minimizing the number of comparisons between the query and the RDF dataset. We also emphasize the role of cost models in the selection of execution plans as well as index sets for a given workload.

We first introduce related approaches for RDF indexing and query processing. We provide an overview of the materialized view selection problem in relational databases and discuss its application for optimization of query processing. Inspired by these techniques, we introduce RDFMatView, a framework for answering SPARQL queries using materialized views as indexes. We provide algorithms to discover those indexes that can be used to process a given query and we develop different strategies to integrate these views in query execution plans. We show that our techniques yield large improvements in processing time, depending on the selected execution plan.

The selection of an efficient execution plan states the topic of our second major contribution. We introduce three different cost models designed for SPARQL query processing with materialized views. A detailed comparison of these models reveals that a model based on index and predicate statistics provides the most accurate cost estimation. We show that selecting an execution plan using this cost model yields a reduction of processing time with several orders of magnitude compared to standard SPARQL query processing.

Finally, we propose a simple yet effective strategy for the materialized view selection problem applied to RDF data. Based on a given workload of SPARQL queries we provide algorithms for selecting a set of indexes that minimizes the workload processing time. We create a candidate index by retrieving all connected components from query patterns. Our evaluation shows that using the set of suggested indexes usually achieves larger runtime savings than other index sets regarding the given workload.

Zusammenfassung

Die Idee des Semantic Web, menschenlesbare verlinkte Inhalte des WWW durch die Anreicherung mit Metadaten in machinenverarbeitbare Informationen zu verwandeln, hat in der Wissenchaftsgemeinde viel Aufmerksamkeit erregt.

Das Resource Description Framework (RDF) wurde vom W3C als ein Datenmodell für semantische Annotationen eingeführt, das Ressourcen und deren Beziehungen in Subjekt–Prädikat–Objekt Tripeln kodiert. Derzeit sind große RDF Datensätze, bestehend aus Millionen von Tripeln, im Web vorhanden. Die Entwicklung von Verfahren, um diese Informationen effizient anzufragen, stellt eine größe Herausforderung dar.

Dazu wurde die deklarative Anfragesprache SPARQL vom W3C entwickelt, um Informationen aus RDF-Daten zu extrahieren. SPARQL–Anfragen werden analog zu select-project-join Anfragen in relationalen Datenbanksystemen formuliert. Allerdings sind Anfragen in großen RDF Datenmengen aufgrund der strukturellen Komplexität der Anfragen sehr zeitaufwändig.

In dieser Arbeit schlagen wir die Verwendung von materialisierten Anfragen als Indexstruktur für RDF-Daten vor. Wir streben eine Reduktion der Bearbeitungszeit durch die Minimierung der Anzahl der Vergleiche zwischen Anfrage und RDF Datenmenge an. Darüberhinaus betonen wir die Rolle von Kostenmodellen und Indizes für die Auswahl eines efizienten Ausführungsplans in Abhängigkeit vom Workload.

Diese Dissertation führt zunächst in verwandten Arbeiten zu RDF Indexierung und Verarbeitung von SPARQL Anfragen ein. Wir geben einen Überblick über das Problem der Auswahl von materialisierten Anfragen in relationalen Datenbanken und diskutieren ihre Anwendung zur Optimierung der Anfrageverarbeitung. Inspiriert durch diesen Techniken stellen wir RDFMatView als Framework für SPARQL-Anfragen vor. RDFMatView benutzt materializierte Anfragen als Indizes und enthalt Algorithmen, um geeignete Indizes fur eine gegebene Anfrage zu finden und sie in Ausführungspläne zu integrieren. Unsere Evaluation zeigt, dass diese Techniken die Bearbeitungszeit einer Anfrage in Abhängigkeit vom gewählten Anfrageplan drastisch reduzieren.

Die Auswahl eines effizienten Ausführungsplan ist das zweite Thema dieser Arbeit. Wir führen drei verschiedene Kostenmodelle für die Verarbeitung von SPARQL Anfragen ein. Ein detaillierter Vergleich der Kostmodelle zeigt, dass ein auf Index– und Prädikat–Statistiken beruhendes Modell die genauesten Informationen liefert, um einen effizienten Ausführungsplan auszuwählen. Die Evaluation zeigt, dass unsere Methode die Anfragebearbeitungszeit im Vergleich zu unoptimierten SPARQL–Anfragen um mehrere Größenordnungen reduziert.

Schließlich schlagen wir eine einfache, aber effektive Strategie für das Problem der Auswahl von materialisierten Anfragen über RDF-Daten vor. Ausgehend von einem bestimmten Workload werden algorithmisch diejenigen Indizes augewählt, die die Bearbeitungszeit des gesamten Workload minimieren sollen. Dann erstellen wir auf der Basis von Anfragemustern eine Menge von Index–Kandidaten und suchen in dieser Menge Zusammenhangskomponenten. Unsere Auswertung zeigt, dass unsere Methode zur Auswahl von Indizes im Vergleich zu anderen, die größten Einsparungen in der Anfragebearbeitungszeit liefert.

Acknowledgments

I would like to express my gratitude to all the people who supported me during the development of this work.

First of all, I would like to thank God. I know you always were, are, and will be there to show us the way we should walk.

I am really grateful to my Supervisor Prof. Dr. Ulf Leser. He gave me the opportunity to be a scientist in his research group. I appreciate very much his detailed and pertinent advices, his support, and invaluable guidance during my research. Apart from him, I thank Prof. Johann-Christoph Freytag, Ph.D. He always took time to discuss and review my work. His valuable questions and comments often showed me the way to refine my ideas.

I am deeply indebted to my family. To my wife Esthela for her understanding and encouragement since the first beginning of this adventure. She always has a fantastic sense of humor, an endless patience and unlimited love that she delivers to each member of our family. My son Roy and my daughter Isabella provided me the strength and motivation to wake up every day and do my best. The best reward was to see those smiling dirty faces at evening waiting for me to come home.

Agradezco a Mamá y Papá que siempre estuvieron y cuidaron de nosotros a pesar de estar al otro lado del mundo. A mis hermanas Addy y Heidi así como a mi sobrina Vale. Ellas influyeron en mi persona más de lo que ellas se imaginan. A la Sra. Esthela Flores, quien viajó desde la Sultana del Norte hasta la Germania (en muchas ocasiones) con tal de ayudarnos y por supuesto, disfrutar a sus nietos, =).

It would not be possible to finish my PhD without the help and support of many friends and colleagues of the WBI group. Special thanks goes to Quang Long Nguyen, Silke Trißl, Andre Koschmieder, Peter Palaga, Samira Jaeger, Astrid Rheinlaender, Johannes Starlinger, Christian Rothe, Hagen Moebius, Ralf Heese, Milan Miladinovic, Maja Repic, Magritt Hoppe, Bernd Palmer, Susanne Tzerlitzke, Victor Uc, Gimer Cervera, Jorge Gomez, and Fernando Curi. There is another multitude of people I should mention here, but doing so, I would require as many pages as this thesis has. Anyway, I thank all of you.

I am grateful to the Consejo Nacional de Ciencia y Tecnología (*Conacyt*) in México and to the Deutscher Akademischer Austausch Dienst (*DAAD*), who provided financial support for my PhD. I thank the detailed evaluation of my research performed by the DAAD and the great technical and administrative service provided by Stephanie Buechl.

Finally, I am particulary grateful to Olaf Hartig. He always had a friendly and open way to share with me his experience working with RDF and SPARQL. I enjoyed the interesting scientific discussions we had during the development of this work.

Contents

1 Introduction

The *Semantic Web* is an initiative that has recently gained considerable momentum [1]. As an evolution of the *World Wide Web*, it aims to create a universal medium for the exchange of data where data can be shared and processed by automated tools as well as by people. The basis to achieve this goal is a logical data model called Resource Description Framework (RDF) [62].

Basically, an RDF dataset is a collection of statements, called *triples*, of the form *(s,p,o)* where s is a subject, p is a predicate and o is an object. Each triple states the relation between subject and object by means of the predicate. A set of triples can be represented as a directed graph where subjects and objects represent nodes and predicates represent edges connecting these nodes[1].

SPARQL is the *W3C* standard query language for searching in RDF datasets [74]. It supports operations similar to select-project-join queries in relational databases. For instance, we can retrieve titles of the articles written by Albert Einstein by applying the SPARQL query in Listing 1 over the dataset described in Table 1.1.

```
SELECT ?title WHERE {
?article <hasTitle> ?title.
?article <hasAuthor> ?author.
?author <hasName> "Albert Einstein".
}
```

Listing 1: SPARQL query to retrieve all articles (co-)authored by Albert Einstein

Table 1.1: Example RDF dataset consisting of six triple statements

subject	**predicate**	**object**
$< article\_1 >$	$< hasTitle >$	"Special Theory of Relativity"
$< article\_1 >$	$< hasAuthor >$	$< Albert\_Einstein >$
$< Albert\_Einstein >$	$< hasName >$	"Albert Einstein"
$< article\_2 >$	$< hasTitle >$	"Distributed Databases"
$< article\_2 >$	$< hasAuthor >$	$< Tamer\_Oszu >$
$< Tamer\_Oszu >$	$< hasName >$	"Tamer Ozsu"

Listing 1 illustrates a simple SPARQL query where each join is denoted by a dot and variables are preceded by "?". The whole *WHERE-CLAUSE* can be seen as a graph pattern that needs to be matched against the data graph represented by the RDF dataset.

[1] Actually, predicate values can also represent subjects and objects. We emphasize this fact in Chapter 2, Definition 2.2

In general, predicates may also contain variables, which increases the complexity of query evaluation.

In this simple example, the values $< article\_1 >$, "Special Theory of Relativity", and $< Albert\_Einstein >$ can be matched to corresponding variables, namely ?*article*, ?*title*, and ?*author*. Note that ?*title* is the only exported variable. Therefore the final result would be: "Special Theory of Relativity".

The increasing amount of RDF data has motivated the development of approaches for efficient RDF data management.

Current SPARQL implementations are built over either relational database technology (for instance PostgreSQL [72], MySQL [66]) or specialized storage systems, (e.g. Jena [92], 3Store [85, 44], Sesame [13]). Other systems have been proposed implementing the common paradigm of a triple table (4Store [45], YARS [46]), often normalized by using two or more tables to store long literal and URI values. In all these systems, when a SPARQL query is executed, the number of joins generated through the processing is roughly the same as the number of patterns in the query. Optimizing these joins is one of the most critical issues to obtain scalable SPARQL systems [86].

Note that even more recent systems, such as RDF-3X [68] or Hexastore [91], are still based on the concept of a singleton triple store, even if they are not built upon relational databases. They reinforce their scalability with customized index structures that help to lookup RDF data at processing time. Nevertheless, the number of required joins remains the same.

This thesis addresses the topic of efficient SPARQL query processing by providing a SPARQL pattern–based indexing method. We analyze in detail the use of materialized views (RDFMatView) to speed up the query processing time. Additionally, we propose a materialized view selection algorithm for RDF data, which suggests an optimal set (under a cost–based criteria) of RDFMatViews to improve the processing time of a given workload of SPARQL queries. Finally, we define different cost models to evaluate RDFMatViews and experimentally test their accuracy.

As mentioned by Jeffrey D. Ullman in the foreword of [6], any index on, or summary of, a database can be seen as a materialized view. Therefore, for simplicity, in the rest of this thesis we use the term *index* and *materialized view* as synonyms.

1.1 Motivating Example: Querying Data with SPARQL

RDF, as a free-schema data model, offers a seamless integration of datasets allowing the exchange and processing of knowledge using automated intelligent methods [53]. On the other hand, due to its inherent graph-structure, querying large RDF datasets requires efficient mechanisms to speed up the retrieval of information.

Examples of huge datasets are, for instance, the UniProt database containing more than 600 million triples [28] or the W3C SWEO Linking Open Data Community with more than 4 billion triples [73]. With such datasets, executing SPARQL queries becomes a problem. Note that the execution time is heavily influenced not only by the size of the dataset (number of triples) but also, as stated before, by the number of joins required to find the

results of the query.

```
SELECT * WHERE {
?s1 ?p1 ?o1 .
?o1 ?p2 "hexokinase" .
?s1 rdf:type ?type1 .
?s1 rdfs:comment ?comment1 .
?s1 rdfs:label ?label1 .
?s1 rdfs:comment ?comment2 .
?s1 rdfs:label ?label2 .
}
```

Listing 2: Example SPARQL query to gather information about Hexokinase enzyme [9]

Example 1.1 (Searching Hexokinase Enzyme) *Consider the query in Listing 2. Executing this query on a conventional SPARQL processor, such as Jena [92] or 3Store [85], results in the computation of six self-joins of a large triple table. However, we can safely assume that types, labels, and comments of an object are used together very often. Therefore, by materializing this information inside the system, the query could be computed with only three joins, as the materialized view would help to retrieve the information on* $s1$*.*

1.2 Context of this Thesis

In this section, we introduce important concepts in the context of our work. We present an overview of research in areas of indexing, SPARQL query processing, and RDF data storage. We also provide an overview of selecting and using materialized views for relational databases. More detailed discussions of closely related work are provided at the end of each chapter.

1.2.1 Semantic Web

The *Semantic Web* (Web of data) envisions an evolution on the current *World Wide Web* where intelligent software agents use machine-readable metadata to access and process information on the Web. It consists of a set of methods and technologies for sharing data, just as the hypertext Web is for sharing documents [8]. Its efforts are led by W3C [7] with participation of a large number of researchers and industrial partners.

The model to describe data in the Semantic Web is the Resource Description Framework (RDF) [62]. However, a number of additional technologies such as data interchange formats (e.g. RDF/XML, N3, Turtle, N-Triples), and metadata notation languages such as RDF Schema (RDFS) [12] and the Web Ontology Language (OWL) [64], are also part of the Semantic Web initiative. By using these resources, the Semantic Web aims to describe concepts, terms, and relationships across sites.

The Semantic Web infrastructure is based on a technology stack, shown in Figure 1.1, consisting of a hierarchy of layers organized such that each layer exploits the functionalities provided by its underlying layers.

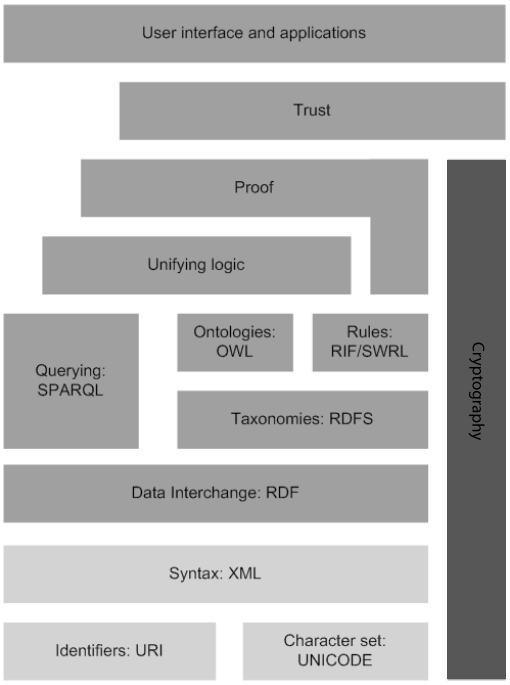


Figure 1.1: Semantic Web Stack. The Semantic Web architecture consists of a set of layers describing technologies to support its vision. Hypertext Web technologies are denoted by light gray boxes and Semantic Web technologies by dark gray boxes. Image redrawn following [49].

The layers can be divided in two main groups:

- Hypertext Web technologies
- Semantic Web technologies

The bottom layer contains Hypertext Web technologies such as *URI* and *UNICODE* syntax. The former allows the unique identification of resources and the latter is a standard to encode, represent, and handle text.

XML is built upon this layer. Basically, XML is used as markup language to create documents with structured data. The use of *XML Namespaces* is also included in this layer to allow markups that reference multiple sources.

On top of Hypertext Web technologies are Semantic Web technologies starting with RDF, a model that allows a schema–free representation of data, RDFS and OWL vocabularies to describe semantics of data, and SPARQL as the language to query and retrieve RDF information.

The upper layers contain several important techniques not yet standardized. They are, however, required to fulfill the complete vision of the Semantic Web [49].

In this thesis, we propose materialized views to store selected semantic data, which can be consumed by a SPARQL query processor to speed up query processing. Our proposal thus, can be located between data interchange and querying layers of the Semantic Web stack.

1.2.2 RDF and SPARQL

The technological basis for Semantic Web applications is a logical data model called Resource Description Framework (RDF) [62]. An RDF dataset is a collection of statements called *triples*, each of the form *(s,p,o)* where *s* is a subject, *p* is a predicate and *o* is an object. Each triple states a relation between *subject* and *object* of the type *predicate*. Any set of triples can be represented as a directed graph where subjects and objects represent nodes and predicates represent edges connecting these nodes.

The SPARQL query language is the W3C standard for querying RDF repositories [74]. The building blocks of a SPARQL query are triple patterns. In essence, triple patterns are RDF triples that may contain variables. Listing 1 shows an example of a SPARQL query containing three triple patterns.

SPARQL query processing consists of graph pattern matching. First, the SPARQL processor receives a query and extracts its *WHERE CLAUSE* that is given as a graph pattern. This graph pattern is matched against the RDF dataset. The query graph pattern matches any subgraph of the RDF data where the RDF terms from that subgraph may be substituted consistently for the variables contained in the query graph. Thus, the results are all subgraphs of the RDF graph that are isomorphic to the query graph.

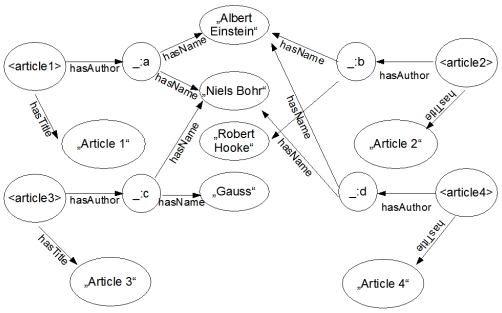


Figure 1.2: RDF data graph containing information about articles and their authors.

Take the SPARQL query provided in Listing 1 (Page 1) and the dataset shown in Figure 1.2. After matching the query pattern against the RDF dataset, we obtain the subgraphs shown in Figure 1.3. The final result set is a projection of the variable *?title* of these subgraphs, i.e., *Article 1*, *Article 2* and *Article 4*.

SPARQL provides different ways to combine triple patterns. These combinations allows the creation of more complex graph patterns:

- A *Basic Graph Pattern* consists of a conjunctive sequence of triple patterns.

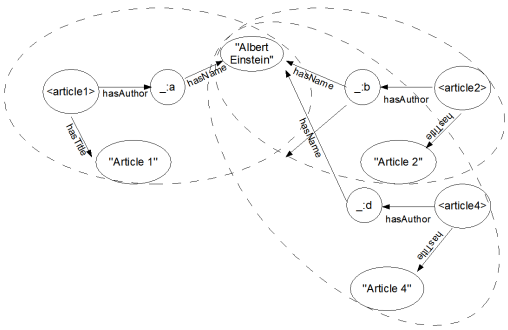


Figure 1.3: Matching subgraphs of the query in Listing 1 over the graph in Figure 1.2.

- An *Optional Graph Pattern* in a SPARQL query consists of a set of triple patterns that may have matches over the RDF data. These matches are a part of the solution of the SPARQL query. However, if no match is found, SPARQL creates no bindings for its contained variables but does not eliminate the solution for the query.
- An *Alternative Graph Pattern* in SPARQL combines graph patterns such that one of several graph patterns may match. In essence it represents the *UNION* operation in set theory. If more than one of the graphs matches, all the possible pattern solutions are found.

In this work, we restrict our approach to SPARQL conjunctive queries, i.e., only queries containing basic graph patterns are allowed. In Chapter 5 we provide directions to include more types of patterns into this approach.

1.2.3 RDF Persistent Data Storage

The increasing amount of RDF datasets on the Web has demanded that most Semantic Web-based applications need a proper RDF storage system as their data backend. To respond to this need, several RDF storage systems have been implemented with different methods and targets. Generally, there are three basic requirements for such systems:

- Efficient query processing
- A standardized query interface
- Support for inference of semantic applications [60]. In short, inference is the process to provide a complete set of results by discovering information that is not explicitly given by either the query or the RDF dataset.

Hertel et al. describe in [50] a generic architecture for RDF storage systems. This architecture consists of two main components: a repository and a logical middleware. The repository stores the data whereas the middleware accesses and manages the data. In

general, the middleware can be described by a set of components categorized by their function, e.g., adding, deleting, and querying data.

Commonly, RDF schemas and instances can be accessed and manipulated in main memory. However, persistent storage of large datasets requires the use of database management systems [92] or customized semantic layouts with a proper schema definition [68].

The simplest generic schema is described by using a so–called triple store. Essentially, this schema consists of one table containing three attributes that represent the components of an RDF triple, i.e., subject, predicate and object. The main advantage of this schema is that every tripe is represented by a single row in the table. Adding new information needs a single insertion in the table. On the other hand, its major drawback comes at query processing as it requires to perform roughly as many joins as the number of patterns contained in the query. In a single table containing a large amount of triple statements this process can be very time consuming.

Several modifications to this schema have been proposed to improve performance or scalability in RDF systems. Harris et al. propose in [85] a *normalized triple store*, storing resources and literal in separate tables. Following this approach, the storage space is significantly reduced, specially when resources and literals are used repeatedly. Wilkinson et al. proposed Jena, a hybrid schema in [92], combining the pure and normalized triple store. They allow to store RDF statements either in a triple or resources table, regarding the extension of the values.

1.2.4 SPARQL Indexing and Query Optimization

Indexing of data and optimization of queries is an active research topic for improving the performance of SPARQL query processing [30, 36, 63, 69, 93]. As the amount of RDF data published on the Web is continuously growing, processing large datasets is a problem, specially when using queries with a large number of patterns. Processing time of answering a SPARQL query over a given dataset is determined mainly by three aspects, i.e. size of the dataset, complexity of the query, and the number of matches of the query in the dataset.

The size of the dataset determines the size of the search space that must be accessed by the query engine to find solutions for a given query. The complexity of a query is basically determined by the number of triple patterns it contains. If a query contains only variables, executing a query with m triple patterns over a dataset with n triples conceptually requires to compare the query pattern with all subgraphs of size equal or less than m in the dataset (potentially each triple pattern may be mapped to the same triple in the dataset).

As triples are typically stored using one or a few tables (triple table approach), answering a SPARQL query consisting of more than one pattern requires the computation of roughly as many joins as the query has patterns. Consequently, optimization of join operations is a critical issue to obtain scalable SPARQL systems [19]. Approaches such as those described in [24, 37, 86, 90], improve query processing by optimizing join operations. The authors positively influence query processing by reordering the query pattern regarding the cardinality of each single pattern. However, the number of joins required to answer the query remains the same.

Contrary to those approaches that concentrate on the relational representation of an RDF storage schema (e.g. [68, 91]), the indexing method that we propose in this thesis aims to fully exploit the RDF graph-structure by considering the occurrence of patterns in other patterns. Note that our indexing strategy is workload dependent. Selected indexes are defined regarding a given workload of SPARQL queries, and therefore, impact only those queries in the given workload. The main advantage of indexing RDF data in a workload independent manner is that its general schema allows to answer any query. We instead index only triple patterns that occur frequently in an expected workload. In this sense, the required space to store these selected indexes is remarkably less than the required space to store each possible combination of RDF terms. Our approach can be considered as a *native RDF/SPARQL indexing* method whose concepts are viable for many implementations of RDF stores. Additionally, it can be seen as an orthogonal indexing method to those strategies indexing RDF terms *(s,p,o)*, and may be used in conjunction with them.

1.2.5 Materialized Views in Relational Systems

A materialized view is a database object that caches the results of a query to provide fast data access [6]. In fact, a materialized view can be seen as a special index structure that may contain fields not only from one table but from a set of tables combined by a specific query. The idea of materialized views has been proposed in the literature decades ago [22]. Problems related to definition, composition, or maintenance of views, especially in relational systems, have been addressed in [11, 15, 20, 39, 54, 77, 83].

Other efforts have been devoted to improve query processing by using materialized views. In essence, these efforts strive to find efficient methods to answer queries minimizing the access to the original data source. This is basically done by generating execution plans containing materialized views that replace parts of the original query [43]. According to Halevy [43], there are mainly two areas in which the problem of answering queries implementing views arises:

- Query optimization and database design
- Data integration

In query optimization, the idea of implementing materialized views to speed–up query processing is suitable, as part of the computation required for the query has already been done and is available in the form of materialized results. The savings achieved may significantly influence at processing time, especially when the results can come from evaluating several query conditions. Using materialized views in query optimization requires to rewrite the query. However, rewriting a query is not a trivial problem. Levy et al. proved in [57] that the problem of finding a query rewriting is closely related to the query containment problem. The authors show that *Containment mappings* [21] provide the core solution to the problem of finding the possible usages of a view for a query.

Different aspects of query processing using persistent results have been addressed in [22, 87, 94]. As mentioned by Chaudhuri et al. [22], the use of materialized views may

improve but also worsen query processing time depending on the query and the statistical properties of the database used to generate the execution plans. Therefore, accurate cost models that help the optimizer to decide which plan to use are indispensable to achieve efficiency in query processing.

Yang et al. describe in [94] a prototype system that rewrites queries by using attribute mappings. Similar to Yang, we propose to rewrite a query using a predefined set of materialized SPARQL queries. However, we deal with the fact that queries may not be fully answered using only materialized results. Thus, the partial results need to be extended to answer the query.

The use of materialized views to speed up queries has also been a topic of interest in commercial relational databases. In that context, Goldstein and Larson proposed in [35] a view matching algorithm to determine whether a materialized view can be used to process a query. The evaluation of this algorithm is based on an implementation in Microsoft SQL Server. The analysis of query and views predicates is performed by using three different tests:

- Equijoin subsumption,
- range subsumption, and
- residual subsumption.

The first test ensures that all columns equal in the view are also equal in the query. The second test verifies that range constraints given in the view are more general than those given in the query. Finally, the third test evaluates those expressions that are neither equalities nor range predicates. Additional to these tests, the authors propose the use of an in–memory index that contains all required information to apply the described tests. This index structure aims to efficiently discard views that can not be used to process a given query.

Further work about views in query processing over commercial database systems is introduced by Zhou et al. in [96]. The objective is to improve query processing by discovering similar subexpressions[2] among a given set of relational queries. To exploit such similarities, a new component, *covering subexpression (CSE) manager*, is added to the query optimizer. The main function of the CSE is to store a special form of metadata, referred here to as *table signature*. Initially, when a query is submitted, the optimizer rewrites the query in different ways. For each unique expression generated by the optimizer, a table signature is computed and registered in the CSE. The table signature describes an expression implementing a binary tuple consisting of a boolean and a list. The boolean indicates whether the the expression contains a group–by operation and the list contains those source tables contained in the given expression. All table signatures are computed only once, stored in memory and can be reused by the query engine at query processing.

To generate candidate CSEs the authors applied a set of heuristic rules that prune out the number of possible candidates. These rules are cost–based and can be described as follows:

[2] An expression is defined as a set of (SPJ) query operators

- A CSE that achieves savings less than a given threshold are disregarded.

  This rule acknowledges the fact that only expensive expressions may significantly influence the overall query processing time. Otherwise, savings in time would be rather minimal to be worth the optimization overhead.

- Candidate CSEs with large results sets are excluded.

  Evidently, a CSE with a large result set produces high materialization and reading costs. At processing time, using an expression to answer a given query would require to read and perform additional computations, such as extending the expressions results to the final query results. In some cases, these operations may be highly time consuming, even higher than computing the query from scratch.

- Generate advantageous combinations of candidate CSEs.

  This rule aims to save redundant computation by merging two candidate CSEs. However, this process is not always beneficial as it may generate a larger result set than than the individual candidates, and in consequence, higher materialization and reading costs.

- Verify CSE containment.

  Equivalent subexpressions can be avoided by checking if a candidate CSE_i is contained by another candidate CSE_j. However, similar to the previous rule, it is important to verify the costs generated when the dominant CSE is computed. For instance, if its set of results is very large it may become less beneficial than the dominated CSE.

Results show that query processing using these heuristic rules and the view matching algorithms provided by Goldstein et al. in [35] can improve. Deciding factors for this improvement can be attributed first, to the pruning process which significantly reduces the optimization overhead, and second, to the reuse of materialized views that allows the query engine to efficiently answer the query.

A second major area where the use of materialized views arises is data integration. Data integration systems, in general, strive to provide a uniform query interface for a set of different data sources. The idea is to provide automatic management of source locations, combination of data, and integration of results to relief the user from the need to have knowledge of each single schema of the different data sources. This is commonly done by using mediated schemas. Mediated schemas are sets of relations designed for specific integration applications. These relations serve as a uniform query interface for all the sources [32]. However, the information is not directly stored in this mediated schema. Instead, a set of semantic mappings between the mediated and local schemas provides the functionality to access the information directly from the data sources.

Lenzerini described in [56] two basic approaches that have been proposed to model the relations between mediated and local schemas. The first approach, called *global–as–view* (GAV), requires a global schema expressed in terms of the data sources. The

second approach, referred to as *local–as–view* (LAV), requires to specify the global schema independently from the sources. The relationships between the global schema and the sources are given by defining every source as a view over the global schema.

Both approaches have advantages and disadvantages. GAV is more beneficial in systems where the set of source tables is stable. This approach favors query processing as it tells the system how to use the sources to retrieve data. The problem comes when a new source is added to the system. The new source may impact on the definition of several elements of the global schema, i.e., it may require to modify the definition of the associated views.

On the other hand, LAV achieves better performance applied on integration systems using a stable global schema. Contrary to GAV, the addition of a new source only requires to create a new view definition that relates the new source with the global schema.

1.2.6 Materialized View Selection Problem

As mentioned in Section 1.2.5, materialized views are an important feature in relational database systems. In this section, we review the problem of selecting views to materialize for a given workload. Those views should be selected that minimize the execution time of the complete workload [29]. This so called MV–selection problem is a non-trivial optimization problem [25], which has been especially studied in relational data warehouses where the large amount of data requires efficient solutions to analyze and generate valuable information [40, 4].

The MV–selection problem can be viewed as a combinatorial optimization problem since its solution has to be chosen from among a finite number of possible configurations. Therefore, explicit and complete enumeration and scoring of all possible MV subsets is, in principle, a possible way to solve it. This method is however impractical in most cases, since the computational effort grows exponentially with the number of candidate MV [25]. For this reason, heuristic algorithms have been proposed to find an approximative solution to the problem, in the sense that they do not return the optimal MV subset, but a "near-optimal" solution [3, 23].

Essentially, heuristics generate a set of candidate views from the workload and evaluate them regarding a cost model to discover which of them achieve larger savings (in time) for the entire workload. In addition, the resulting set of views usually must satisfy certain constraint or set of constraints about a resource (e.g. disk space or number of views). Finally, a set of views is selected and materialized in a given repository. A detailed description of these steps is provided later in Chapter 4.

1.3 Contributions

In this thesis, we study the topic of answering queries using materialized views with RDF and SPARQL. Our goal is to provide a framework to improve SPARQL query processing over large RDF datasets using materialized queries. Besides integrating materialized views into SPARQL query processing, we also report on results regarding a materialized view selection approach. The evaluation of views, queries and workloads is based on a novel

statistical cost model for RDF data.

Specific contributions of our research are:

- Implementation of an approach for answering SPARQL queries using *materialized views* for RDF datasets (RDFMatView). The implementation includes algorithms to optimize the selection of a set of materialized views from a given search space, rewriting strategies to generate different execution plans to answer a query and a method to materialize SPARQL queries using relational systems. We integrate our solution into the SPARQL Jena framework.
- Analysis and development of cost models to evaluate query execution plans. The use of cost models strives for an efficent selection of a query execution plan regarding statistical measures of the data. We provide an in-depth analysis and a thorough evaluation of three different models. All models are implemented and integrated into RDFMatView.
- We provide a simple yet effective methodology that, from a workload of SPARQL queries, suggests an optimal set of SPARQL materialized views to improve the processing time of the workload. We propose a strategy to generate candidate for materialized views from connected components of the query patterns in the workload and analyze their eligibility for each query. Additionally, a cost model to evaluate all materialized views and their influence on query processing is provided. The cost model estimates the time reduction that the usage of a materialized view may achieve over the given workload.

1.4 Structure

This thesis is structured as follows:

- Chapter 2 gives an overview of our approach using materialized views for RDF data and their integration in SPARQL query processing. We provide basic concepts and algorithms.
- In Chapter 3, we propose and analyze three different cost models to evaluate materialized views and its relevance in SPARQL query processing.
- In Chapter 4 we describe our proposal for a materialized views selection approach for RDF data.
- Finally, Chapter 5 summarizes our findings and highlights future research directions.

1.5 Relation to Prior Work

Chapter 2 of this thesis describes RDFMatView. Section 2.2 describes the logical part of this framework initially proposed by Rothe [76] and described by Heese et al. in [48].

Sections 2.3 and 2.4 are extension of the original approach presented by Castillo et al. in [18, 19]. The contributions described in [18, 19] can be attributed to the authors as follows: Leser conceived and supervised the project. Rothe set the fundamental concepts of the system and implemented a prototype. Castillo proposed and developed a method to materialize SPARQL queries and three strategies to rewrite SPARQL queries using indexes. All evaluations were performed by Castillo. Leser and Castillo wrote the publications.

Chapter 3 is devoted to describe three cost models for query optimization. The first model (SyCoM, Section 3.2) was initially proposed by Heese et al. in [48]. The second model (CardiOS, Section 3.3) was proposed by Moebius in [65] and supervised by Castillo and Leser. Castillo refined this model by providing further analysis of cycles in Section 3.3.4, proposed a third novel model (SPOracle, Section 3.4), and performed all evaluations. Leser supervised and suggested improvements for all models.

Chapter 4 is based on the ideas introduced by Castillo and Leser in [17] regarding materialized view selection. Leser defined the research directions. Castillo proposed the algorithm to suggest an optimal set of materialized views described in Section 4.1, proposed a cost model to evaluate the potential views, implemented the system, and performed all evaluations.

2 RDFMatView: Concept and Query Rewriting

In the previous chapter we illustrated the need of an efficient RDF data management. We mentioned related approaches, which aim to improve RDF storage (Section 1.2.3), index RDF data (Section 1.2.4), and sketched the use of materialized queries to speed up data retrieval (Section 1.2.5). In this chapter, we described our approach built upon the ideas proposed by Heese et al. in [48] regarding a theoretical framework to answer queries using materialized views. We extend this approach in several parts [19]. First, we describe how it can be integrated into an existing SPARQL query processor. This integration touches several components of a system: We need to be able to execute SPARQL queries and to store their results (plus some metadata) persistently. Second, to use indexes in query processing, we need to intercept the query processor to, at the right point in time, search for an optimal execution plan. Additionally, we must change the way how queries are executed. The query pattern must be divided into that part that is executed by using materialized results and the rest of the query patterns that are not possible or not beneficial to execute using them.

In the remain of this thesis, we refer to materialized views as RDFMatView indexes. We first formally introduce all necessary concepts for this idea in Section 2.1. Section 2.2 defines materialized queries as indexes and shows how one can decide which of a set of specific indexes is suitable for a given query. Those indexes are called eligible, and a set of eligible indexes may be combined to cover a query completely or partly. Section 2.3 describes the algorithm which produces all possible covers for a query given a set of indexes. An extensive example to better understanding of our ideas is also presented in Section 2.3. Afterwards, in Section 2.4 we provide a detailed description of three rewriting strategies that we developed to integrate RDFMatView into a SPARQL processor. We show the results of our evaluation in Section 2.5 and conclude this chapter in Section 2.6 by providing a discussion of current related work.

2.1 Preliminaries

In Chapter 1 we provided a brief description of RDF and SPARQL. Now, we describe in detail their basic elements such as RDF-Term, Triple, Dataset and Patterns.

RDF is built upon the notion of triples. Intuitively, an RDF graph is a set of triples, which in turn consist of RDF Terms referred to as *subject, predicate and object*. An RDF Term consists of an *Internationalized Resource Identifier* (IRI), blank node or an RDF Literal. The set of all RDF terms is denoted as *RDF-T*.

SPARQL is the query language proposed by the *W3C*[1] for retrieving information from RDF graphs [74]. The building blocks of a SPARQL query are triple patterns. Triple patterns are basically RDF triples that can contain variables. A set of triple patterns, in turn, form a basic graph pattern. In the rest of this thesis we will refer to basic graph pattern as pattern. Now, we proceed to formally define these concepts.

Definition 2.1 (RDF-Term, Variables) *Let I be the set of* IRI*s,* RDF-L *the set of RDF-Literals and* RDF-B *the set of anonymous nodes (blank nodes).*
Then, the set RDF-T *of RDF-Terms is defined as*

$$\textit{RDF-T} = I \cup \textit{RDF-L} \cup \textit{RDF-B}.$$

The set V of variables is infinite and disjoint from RDF-T. □

Definition 2.2 (RDF-Triple, Dataset) *Let*

$$D := (I \cup \textit{RDF-B}) \times I \times \textit{RDF-T}$$

be the set of possible RDF-Triples*. Then, any $G \subseteq D$ is called a* Dataset*.*
For a dataset G

$$V(G) := \{v \in \textit{RDF-T} \,|\, \exists (v, x, y) \in G \vee \exists (x, y, v) \in G\}$$

is the set of nodes of G and

$$E(G) := \{(s, p, o) \,|\, s, o \in V(G), p \in I\}$$

the set of edges of G, where p is the predicate that defines a relation between the subject s and the object o. □

According to Definition 2.2, a predicate may be defined from the sets of subjects and objects elements. This is possible because the domains intersect each other, i.e., $I \subset$ RDF-T. However, in this thesis, we restrict ourselves to datasets, where any predicate from one triple cannot occur at subject or object position in other triples. This restriction allows us to visualize an RDF dataset as a graph where subjects and objects are represented as nodes, whereas the predicates are exclusively represented as directed, labelled edges.

Definition 2.3 (Triple pattern, Basic Graph Pattern) *Let T be a set of Triple Patterns defined as*

$$T := (I \cup V) \times (I \cup V) \times (\textit{RDF-T} \cup V).$$

Then any $t \in T$ is called a Triple Pattern*. A* Basic Graph Pattern *P is a set of triple patterns, i.e. $P \subseteq T$.* □

[1] www.w3.org

2.2 Patterns, Occurrences and Indexes

In Section 2.1, we described the basic elements of RDF and SPARQL. At the beginning of this chapter, we also roughly mentioned concepts such as indexes, patterns, mappings and occurrences that make up the RDFMatView approach. We turn now to define them in detail.

RDFMatView has been conceived as an indexing strategy to speed-up SPARQL queries. Indexes in RDFMatView are represented as materialized SPARQL queries, which regarding a set of triple patterns and their matches over a dataset allow the reuse of this information to answer other SPARQL queries.

When matching a pattern against a given RDF dataset we look for a subgraph such that when RDF-Terms from that subgraph are bound to the pattern variables, the result is an RDF graph equivalent to that subgraph. In RDFMatView, we generalize this match process to include variables as possible elements to substitute other variables and not only RDF-Terms. This generalization is required so that we are able to substitute query for materialized patterns. Such substitution is performed by using mappings between index and query patterns.

Before we can define what we consider as an index over RDF graphs we explain concepts our indexes are built upon such as query pattern, mapping and occurrence of a pattern. Definitions 2.5, 2.6 and 2.7 describe them respectively.

In accordance to Definition 2.3, a triple pattern may be visualized using graphs. Figure 2.1 shows the graph representation of a triple pattern.

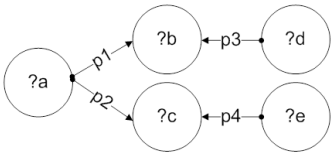


Figure 2.1: Graph representation of a triple pattern

Similar to SQL queries in relational databases, SPARQL queries consist of two main parts: The head that contains a set of exported variables, and the body that represents the conditions given by a set of triple patterns. Definitions 2.4 and 2.5 provide a description of SPARQL query and SPARQL query pattern respectively. Note that our definition of SPARQL query is given regarding the restriction described in Definition 2.2 .

Definition 2.4 (SPARQL query) *A SPARQL query Q consists of a set of exported variables v and a basic graph pattern P where:*

- *P does not contain variables in the places of predicates.*
- *In P, values used in places of predicates do not occur in places of subjects or objects.*

Definition 2.5 (Query Pattern) *Let Q be a SPARQL query. Then, $P(Q)$ denotes its* query pattern*, which is the set of triple patterns in the body of Q.* □

Definition 2.6 (Mapping, total Mapping) *Let P be a query pattern and V_P the set of variables in P. A mapping is a function defined as follows:*

$$S : V_P \rightarrow \textit{RDF-T} \cup V$$

If $S(v) \notin V$ for all $v \in V_P$, then S is a total mapping. □

Our notion of mapping and total mapping is based on the SPARQL-Standard [74] and its definition of *pattern solutions*. However, we want to emphasize that while in the SPARQL standard such solutions are only searched in the data graph, we also permit that variables are mapped to other variables. This generalization allows us to search occurrences of patterns in other patterns, in particular, occurrences of indexes in a query.

Definition 2.7 (Occurrences of a pattern) *Let P_1 and P_2 be two query patterns. P_1 occurs in P_2, denoted by $P_1 \sqsubseteq P_2$, iff there is a mapping S such that $S(P_1) \subseteq P_2$. Such S is called an* embedding *of P_1 in P_2.* □

When we speak about a concrete occurrence of an index pattern in a query pattern, we will refer it to as an *embedding* (to contrast from the term occurrences, which we from now on only use for matches of a pattern in the data graph). Figure 2.2 illustrates an embedding of a pattern P_1 in a pattern P_2.

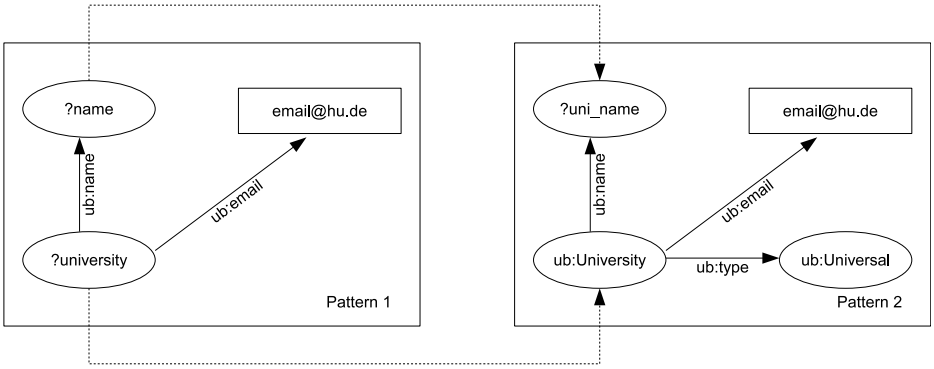


Figure 2.2: *Pattern* 1 occurs in *Pattern* 2 using mapping: $?name \Rightarrow ?uni\_name$ and $?university \Rightarrow ub : University$. Clearly, fixed values must also coincide with each other $ub : name \Rightarrow ub : name$ and $ub : email \Rightarrow ub : email$ to complete the matching between both patterns.

Using the previously introduced concepts, we can now define an index over an RDF data graph.

Definition 2.8 (Indexes) *An* index *I over a RDF data graph G is a pair $I = (P, O)$, where P represents a query pattern and O represents the set of all occurrences of P in G. P is called the index pattern of I. For an index $I = (P, O)$, the* size of *I, written as $|P|$,*

is the number of triple patterns in P. The frequency of I in G, written as $\#_G(P)$, is the number of occurrences of P in G. □

These notations are used for both indexes and queries, i.e., $|Q| := |P(Q)|$ is the *size of a query* Q and $\#_G(P(Q))$ is its frequency. From here on, frequency will be denoted as $\#(P)$, assuming that query and data graph are clear from context.

As RDFMatView strives to speed-up SPARQL query processing by using other materialized SPARQL queries as indexes, it differentiates two main processes to fulfill this task: index creation and query processing. During index creation, indexes are computed and persistently materialized and the results are made available for the query processor. In our approach, indexes are defined offline, i.e., created by an administrator before queries are executed. On the other hand, at query time, the system needs to determine which of the existing indexes are useful for the given query Q. Clearly, only those indexes are candidates for speeding up Q whose patterns are contained in $P(Q)$, i.e., indexes which have an embedding in Q. We call all such indexes *eligible for* Q. Example 2.9 gives an example of eligible indexes for a given query.

Example 2.9 (Suitable Indexes for a Given Query) *Let q be a query pattern and let i_1 and i_2 be two index patterns where $q = \{(?a, p1, ?b), (a?, p2, ?c)\}$, $i_1 = \{(?x, p1, ?y)\}$ and, $i_2 = \{(?x, p2, ?y), (?y, p3, ?z)\}$. Let D be an RDF dataset described in Table 2.1.*

Table 2.1: Tabular representation of an RDF dataset.

Subject	**Predicate**	**Object**
n1	p1	n2
n1	p2	n3
n3	p3	n4
n5	p1	n6
n5	p2	n7
n8	p1	n9
n8	p2	n10
n10	p3	n11

According to Definition 2.7, i_1 has an embedding in q by mapping $?x \Rightarrow ?a$ and $?y \Rightarrow ?b$. In this sense, each occurrence of the triple pattern $(?a, p1, ?b)$ of q must be contained in the set of occurrences of i_1.

Matching i_1 over the dataset D described in Table 2.1 results in a set containing all subgraphs in D equivalent to i_1.

$$\{(n1, p1, n2), (n5, p1, n6), (n8, p1, n9)\}$$

.

Similarly, computing the set of occurrences of q over the dataset D, results in the following set:

$$\{\{(n1, p1, n2), (n1, p2, n3)\}, \{(n5, p1, n6), (n5, p2, n7)\}, \{(n8, p1, n9), (n8, p2, n10)\}\}$$

.

Notice that, each solution of the mapped variables of q appears in the set of occurrences of i_1. To get the final solutions of q, it is required to compute the pattern $(?a, p2, ?c)$ and join the solutions with the results of i_1. Therefore, i_1 is an index suitable for q.

In a similar fashion, analyzing i_2, there exists no mapping between the patterns, i.e., it is not possible to map variables ?y and z from i_2 to any variable of q. Further, the set of solutions for i_2

$$\{\{(n1, p2, n3), (n3, p3, n4)\}, \{(n8, p2, n10), (n10, p3, n11)\}\}$$

evidences that regarding the number of solutions, the patterns of i_2 are even more restrictive that the patterns of q. This fact makes impossible to extend these results to results of q.

Following these arguments, we conclude that i_2 can not be used in the processing of q.

The following definition formally captures this idea.

Definition 2.10 (Eligible Index) *Let G be a data graph, $\mathfrak{I}$ the set of indexes on G, and Q a query against G. We call an index $I \in \mathfrak{I}$ eligible for Q iff $P(I) \sqsubseteq P(Q)$. The set of all eligible indexes for query Q is denoted by $\mathfrak{I}_Q$.* □

Definition 2.10 describes which indexes can be used to help processing a given query. Note that an eligible index can be used in different ways to process a query if it has different embeddings.

Using a combination of indexes for a given query may bring even more advantages at processing time. The more eligible indexes are used to process a query, the larger the number of query patterns are substituted by index preprocessed information. In other words, replaced query patterns do not need to be matched against the RDF dataset, because this process has been done in advanced when indexes were computed. However, partial results coming from these indexes do need to be joined to extend their results and generate the results of the query. Such join operation must be made, in the best situation, over overlapping relations between indexes, or, failing that, combining index results by complete joining operations *(Cartesian Product)*.

Therefore, overlapping indexes are good candidates for reducing query processing time because the query engine can combine occurrences of these indexes and thus quickly generate solutions for larger fractions of the query pattern.

We define two ways in which indexes can overlap. Two indexes overlap intensionally iff there *could* exist a triple pattern in which their materialization would overlap. In contrast, two indexes overlap extensionally if their materializations overlap on a concrete data graph. Thus, intensional overlap relies only on the pattern of indexes and is independent of a concrete data graph, while extensional overlap needs to consider the actual data graph.

Definition 2.11 (Overlapping Indexes) *Let $I_1 = (P_1, O_1)$ and $I_2 = (P_2, O_2)$ be two indexes over a data graph G.*

- *I_1 and I_2 intensionally overlap iff there exists mapping functions S_1, S_2 such that*

$$S_1(P_1) \cap S_2(P_2) \neq \emptyset$$

- *I_1 and I_2 extensionally overlap in G iff*

$$O_1 \cap O_2 \neq \emptyset$$

□

However, when we want to use overlapping indexes for processing of a query Q, we need to refine our definitions as the query strongly restricts the mappings we need to consider.

Definition 2.12 (Overlapping Embeddings) *Let $I_1 = (P_1, O_1)$ and $I_2 = (P_2, O_2)$ be two indexes over a data graph G. Let Q be a query over G with $P_1 \sqsubseteq Q$ and $P_2 \sqsubseteq Q$, and let m_1 be an embedding of P_1 in Q and m_2 an embedding of P_2 in Q[2].*

- *m_1 and m_2 intentionally overlap in Q iff*

$$m_1(P_1) \cap m_2(P_2) \neq \emptyset$$

- *m_1 and m_2 extensionally overlap in Q and G iff*

$$m_1(O_1) \cap m_2(O_2) \neq \emptyset$$

□

Computing intensional overlaps can be implemented efficiently as this property is independent from the actual data graph (and updates to it). In contrast, computing extensional overlaps is costly as it requires execution of index queries and comparison of their results on a given data graph. On the other hand, at query execution time information about extensional overlaps would be more important than those about intensional overlaps, as the latter is only a necessary yet not sufficient condition for the existence of a concrete overlap given the query. Actually, if two embeddings intensionally overlap in the query but do not extensionally overlap in the data graph, one can immediately conclude that the query has no answer. However, for the rest of this thesis we only consider intensional overlaps to avoid the costly pre-computation and maintenance of extensional overlaps. Example 2.13 shows the behavior of two embeddings, which intensionally overlap but do not overlap at all in the dataset (extensional overlapping).

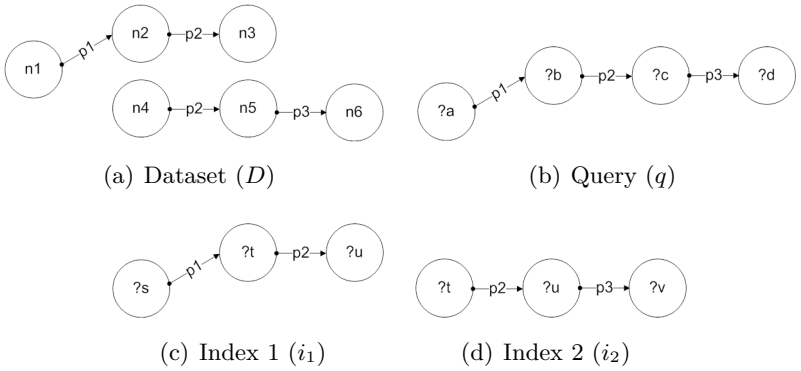
[2]With slight abuse of notation. By $m_1(O_1)$ we mean the projection of all occurrences in O_1 using m_1.

Example 2.13 (Intensional vs. Extensional Overlapping) *Let q be a query pattern and let i_1 and i_2 be two index patterns where $q = \{(?a, p1, ?b), (b?, p2, ?c), (?c, p3, ?d)\}$, $i_1 = \{(?s, p1, ?t), (?t, p2, ?u)\}$ and, $i_2 = \{(?t, p2, ?u), (?u, p3, ?v)\}$. Let D be an RDF dataset described in Table 2.2.*

Table 2.2: Tabular representation of an RDF dataset.

Subject	Predicate	Object
n1	p1	n2
n2	p2	n3
n4	p2	n5
n5	p3	n6

i_1 and i_2 intensionally overlap in the query over the pattern $((?t, p2, ?u))$. Nevertheless, to find out whether the original query q has occurrences on D, one muss obtain and join the results from both indexes. On the other hand, by means of extensional overlapping validation between the same indexes can be proved that there exists no triple in this dataset, where i_1 and i_2 intersect. Therefore, we could immediately infer that q has no possible answers on D. The analysis of this example concludes that extensional overlapping offers more accurate information about the query than intensional overlapping properties. Figure 2.3 shows a graphical representation of this example.



(a) Dataset (D) (b) Query (q)

(c) Index 1 (i_1) (d) Index 2 (i_2)

Figure 2.3: Intensional and extensional ovelapping. i_1 and i_2 intensionally overlap in q over the pattern $(?t, p2, ?u)$. Nevertheless, they do not extensionally overlap over the dataset D.

Using the notion of overlaps, we can finally define the *cover of a query.*

Definition 2.14 (Cover) *Let Q be a query and E_Q the set of all embeddings of eligible indexes for Q in Q. Let $C \subseteq E_Q$. Build a graph G_C for C as follows: Each embedding in C is represented as a node. Whenever two embeddings from C intensionally overlap in Q,*

we add an edge to G_C between the nodes representing the embeddings. Any C for which G_C has only one connected component is called a cover for Q. □

We focus on covers with overlapping embeddings since they allow better estimations of the cost savings that can be achieved with them (see Section 2.3).

Furthermore, we are only interested in maximal covers, i.e., those covers which cannot be extended further by adding new embeddings. Figure 2.4 shows an example of indexes with overlapping and non-overlapping embeddings.

Definition 2.15 (Maximal Covers) *Let Q be a query and C_1 , C_2 be two covers for Q. C_1 is subsumed by C_2 if $C_1 \subseteq C_2$. Any cover which is not subsumed by another cover is called* maximal. □

In the following, we only consider maximal covers. We classify those into two different groups:

- A cover is complete if it covers *all patterns* of a query.
- A cover is partial if it is not complete.

Usually it is not possible to find a complete cover of a query. According to this, we refer in the rest of this thesis to a partial cover as a cover.

2.3 Algorithm for Finding Covers

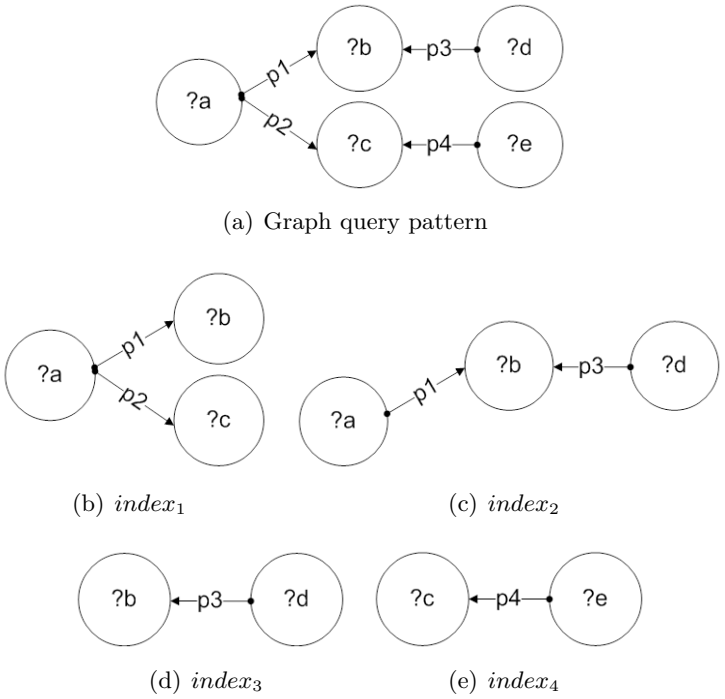
Definition 2.14 is purely conceptual. We now show how we actually compute the set of eligible indexes and how we combine their embeddings to find all covers.

The first task can be solved by classical algorithms for query containment of relational queries [43]. The *Query Containment* problem states that given two conjunctive queries q_1 and q_2, q_1 is contained in q_2, i.e., $q_1 \subseteq q_2$, when for each database instance the results of q_1 are contained in the results of q_2.

Recall that SPARQL queries can be seen as relational queries, which access multiple times (as many times as triples in the query pattern) a large single triple table and therefore, query containment algorithms can be applied. The schema of this table consists of basically three attributes, namely subject, predicate and object. Thus, exported variables of a SPARQL query, can be seen as the head of a Datalog query, whereas the triple patterns and the rest of the conditions, as the body [71, 79]. Listing 3 shows the Datalog representation of the SPARQL query described in Listing 1. In this case, variable *title* is exported in the head of the query, the body of the query consist of the predicate *triple* with the variables *article, title, author* and constants values *hasTitle, hasAuthor, hasName and Albert Einstein*.

```
q(title):-triple(article, 'hasTitle', title),
          triple(article, 'hasAuthor', author),
          triple(author, 'hasName', 'Albert Einstein').
```

Listing 3: Datalog representation of the SPARQL query described in Listing 1.



(a) Graph query pattern

(b) $index_1$

(c) $index_2$

(d) $index_3$

(e) $index_4$

Figure 2.4: Query patterns and indexes with overlapping and non-overlapping embeddings on the query pattern. $index_1$ and $index_2$ overlap in the query pattern *(?a,p1,?b)*. $index_3$ and $index_4$ do not overlap in the query pattern.

Essentially, we find all possible mappings between any index pattern and the query pattern by enumerating and testing all mappings. If a valid mapping exists then we can conclude that an index is eligible for that query and we store the mapping as an embedding. Note that, for a given index, there are potentially many different ways to be eligible, i.e., different mappings between index and query patterns and therefore, multiple embeddings. Algorithm 1 illustrates this process.

The core of Algorithm 1 is located at Line 5, where all embeddings of an index in the query pattern are computed. The implementation of this step is shown in Algorithm 2, which traverses a tree representing the search space of all possible mappings from the index into the query.

Each level in the tree contains all mappings for a specific triple pattern. All mappings of a level in the tree are children of each mapping of the previous level. In Line 12 we generate this tree and traverse it using backtracking in Line 13. During the traversal, the mappings for the different triples are combined (if compatible) to increasingly larger mappings. Whenever all triples of the index have been mapped to the query pattern by one mapping, this mapping is added to the set of embeddings. The complete traversal of the tree is shown in Algorithm 3.

During traversal of the tree, partial index occurrences are successively extended to

Algorithm 1 testContainment. Pseudo code for SPARQL query containment. The algorithm computes all embeddings of indexes in a given query.

Given: Query Q, set of indexes $\mathfrak{I}$
Search: Set E_Q of all Embeddings

```
1: P(I), P(Q) {index and query patterns}
2: 𝔒 := ∅ {Set of embeddings of P(I) in P(Q)}
3: E_Q := ∅ {Set of all embeddings}
4: for all Index I in 𝔍 do
5:    𝔒 := {S|S(P(I)) = P(Q)}
6:    if 𝔒 <> ∅ then
7:       E_Q = E_Q ∪ 𝔒
8:    end if
9: end for
```

occurrences of the of the contiguous triple patterns or eliminated when an extension is not possible (by means of a mapping). Through this strategy, the partial occurrences grow step by step to occurrences of the complete pattern $P(I)$ creating an embedding or they are eliminated during the process. Notice that an index pattern may have more than one embedding on the query pattern, according with the number of mappings that can be found during the traversal of the tree.

For better understanding of Algorithm 3 we show the traversal of a tree in Example 2.16.

Example 2.16 (Finding embeddings of and index in a query) *Let i be an index and q be a query pattern where:*

```
i = {(?a, p_1, ?b),(?b, p_2, ?c)}

q={(?x, p_1, ?y),(?x, p_2, ?z),(?y, p_2, ?z)}
```

Algorithm 2 generates a tree where each level contains all possible mappings between an specific index triple pattern and the query. Figure 2.5 shows a graphical representation of this tree.

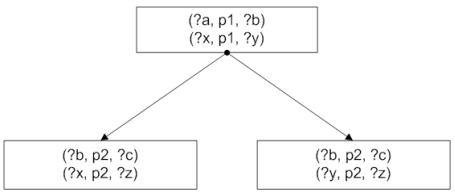


Figure 2.5: Example of a mapping tree.

Starting the traversal from the root node, Algorithm 3 has stored the initial mapping (?a ⇒?x, ?b ⇒?y). The second step goes down to the most–left node and verifies if the

Algorithm 2 searchEmbeddings. Pseudo code for searching all embeddings of an index in a query patterns. It maps variables from the index to the query

Given: Query pattern $P(Q)$, index pattern $P(I)$
Search: $\mathfrak{O}$, set of embeddings of $P(I)$ in $P(Q)$

```
1: L_t := ∅ {Temporal list of occurrences of each index triple pattern in P(Q)}
2: L := ∅ {Final list of t_i occurrences in P(Q)}
3: for all Triple Pattern t_i in P(I) do
4:    for all Triple Pattern t_q in P(Q) do
5:       if  t_i occurs in t_q with mapping S then
6:          L_t := L_t ∪ S
7:       end if
8:    end for
9:    L := L ∪ L_t
10:   L_t := ∅
11: end for
12: occTree := createTree(L)
13: return  𝔒 := traverse_tree(occTree.root)
```

Algorithm 3 traverseTree. Pseudo code for traversing a tree searching for embeddings between an index and a query pattern. The algorithm traverses a tree using backtracking in a depth-first manner.

Given: *element*, {root element of a tree of occurrences of each index triple pattern in $P(Q)$}
Search: $\mathfrak{O}$, set of embeddings of $P(I)$ in $P(Q)$

```
1: comp := false, m_t := ∅, 𝔒 := ∅ , {Boolean, temporal and total lists of embeddings of
   P(I) in P(Q)}
2: if  element.getParent() <> null  then
3:    comp := comparemapping(element, element.getParent()) {Compare mappings}
4: else
5:    comp := false
6: end if
7: if comp = true then
8:    m_t.add(element) {Extend mappings if compatible}
9: end if
10: if comp and element.getChildren <> null then
11:   𝔒.add(m_t) {Add embedding if extended and complete}
12: end if
13: next := element.getchildren() {Verify existence of next level}
14: while next <> null do
15:   traverse_tree(next) {Perform backtracking of T}
16:   next := next.getSibling() {Traverse the occurrences of one level}
17: end while
18: return  𝔒
```

mapping (?b ⇒?x, ?c ⇒?z) of this node is compatible to the mapping already stored (line 3). Evidently, ?b is already mapped to ?y and therefore, mapping ?b ⇒?x is not compatible. We perform backtracking and proceed with the sibling of this node (line 16). Note that if the index would contain more patterns, the branch under this node could be immediately pruned.

Algorithm 3 continues traversing the tree at the sibling right node. Again, we verify the compatibility between mappings (?b ⇒?y, ?c ⇒?z) and (?a ⇒?x, ?b ⇒?y). At this point, we are at the leaf level and the mappings are compatible. Thus, we add the new mapping e.g. ?c ⇒?z and determine that i has an embedding in q. By means of this embedding, i could be used in the processing of q.

Having all embeddings, we proceed to generate covers. We first compute all intensional overlaps between indexes and store them in a matrix. We then incrementally build maximal covers by finding all maximal connected components in this matrix.

Example

We illustrate all previously introduced concepts using a comprehensive example. Consider the RDF data listed in Figure 2.6, the SPARQL query Q_1 shown in Listing 4, and the two RDFMatView indexes I_1, I_2 from Listing 5 and 6.

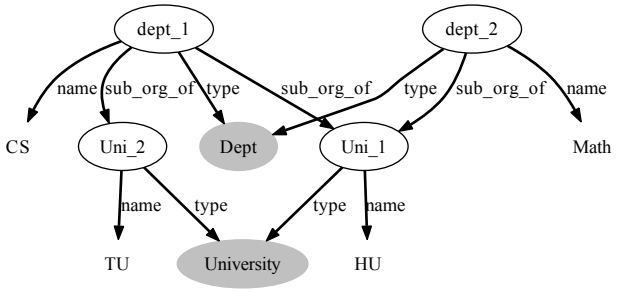


Figure 2.6: RDF dataset containing two universities and their departments.

```
SELECT * WHERE {
?university   rdf:type ub:University;
     ub:name ?university_name.
?ub_department rdf:type ub:Department;
     ub:name ?ub_name_department;
     ub:subOrganizationOf ?university .
}
```

Listing 4: SPARQL query computing universities and their departments

```
SELECT * WHERE {
     ?place   rdf:type ?place_type;
          ub:name ?place_name.
```

```
}
```

Listing 5: RDFMatview index computing places and their names

```
SELECT * WHERE {
?ub_department rdf:type ub:Department;
    ub:name ?ub_name_department;
    ub:subOrganizationOf ?university;
?university  rdf:type ub:University.
}
```

Listing 6: RDFMatview index computing universities with their departments

Executing Q_1 on the dataset in Figure 2.6 produces the result shown in Table 2.3.

Table 2.3: Result of Q_1.

Query Result Set			
university	**university_name**	**ub_department**	**ub_name_department**
Uni_1	HU	dep_1	CS
Uni_1	HU	dep_2	Math
Uni_2	TU	dep_1	CS

Materializing I_1 and I_2 produces results as given in Table 2.4 and Table 2.5, respectively.

Table 2.4: Result of I_1.

Index1		
place	**place_type**	**place_name**
Uni_1	University	HU
Uni_2	University	TU
dep_1	Dept	CS
dep_2	Dept	Math
dep_1	Dept	CS

Table 2.5: Result of I_2.

Index2		
ub_department	**university**	**ub_name_department**
dep_1	Uni_1	CS
dep_2	Uni_1	Math
dep_1	Uni_2	CS

Both I_1 and I_2 are eligible for the query. Actually, I_1 is eligible in two different ways, as it may either substitute the link between a university and its name or the link between

a department and its name. The three resulting embeddings are shown in Table 2.6 and Table 2.7.

Table 2.6: Embeddings of I_1 in Q_1.

Index		Query
Embedding 1		
?place	⇒	?university
?type	⇒	ub:University
?place_name	⇒	?university_name
Embedding 2		
?place	⇒	?ub_department
?type	⇒	ub:Department
?place_name	⇒	?ub_name_department

Table 2.7: Embeddings of I_2 in Q_1.

Index		Query
?ub_department	⇒	?ub_department
?ub_name_department	⇒	?ub_name_department
?university	⇒	?university

The first embedding of index I_1 intensionally overlaps with the embedding of index I_2 in the triple pattern *?university rdf:type ub:University*. Thus, these two embeddings form a cover (in this case the only cover with more than one embedding).

Assume for now the RDF data would be stored in a RDBMS within a single *triple* table, for instance, *Triple(subj, prop, obj)*. Hence, the query in Listing 4 could be answered by the SQL query described in Listing 7.

```
SELECT t1.subj AS a0, t2.obj AS a1, t3.subj AS a2, t4.obj AS a3
FROM Triple AS t1, Triple AS t2, Triple AS t3,
     Triple AS t4, Triple AS t5
WHERE t1.prop= 'type' AND t1.obj= 'University' AND
     t2.prop= 'name' AND t3.prop= 'type' AND
     t3.obj= 'Department' AND t4.prop= 'name' AND
     t5.prop= 'subOrganizationOf' AND
     t1.subj = t2.subj AND t3.subj = t4.subj AND
     t3.subj = t5.subj AND t1.subj = t5.obj;
```

Listing 7: SQL to answer the query from Listing 4

Listing 7 shows that four self joins are required. However, using the pre-computed data for Index1 and Index2, one can answer the query with only one join, as shown in Listing 8.

```
SELECT index2.university,
    index1.place_name AS university_name,
    index2.ub_department,
    index2.ub_name_department
FROM index1, index2
WHERE index1.place = index2.university;
```

Listing 8: SQL representation of SPARQL query in Listing 4 using RDFMatView indexes

This example illustrates that it may make sense to use materialized queries as indexes to process SPARQL queries. Note that in this special example we can actually answer the query only from the materialized indexes. In fact, we do not need to access the RDF database at all because the query pattern can be covered. However, is not always possible to cover a query using only indexes. In these cases the results of a cover must be combined properly with those resulting from querying parts of the query that are left uncovered against the RDF database.

2.4 SPARQL Query Rewriting: Adding RDFMatView to SPARQL Queries

In this section, we describe how our RDFMatView approach can be integrated into an existing SPARQL query processor. Such an integration touches upon several components of a system: First, we need to be able to execute selected queries and to store their results (plus some metadata) persistently. To use indexes in query processing, we need to intercept the query processor to search for an optimal cover. Finally, we must change the way how queries are executed, as we need to divide the query pattern into that part that is covered by the chosen cover - which is answered by retrieval of the materialized information - and the rest of the query pattern. We present solutions to all these steps for the ARQ system (SPARQL query engine) of the Jena framework [5]. However, we want to stress that the general process would be the same for any other SPARQL query processor. Our decision is based on the modular architecture of Jena that allows a seamless integration of additional functionality. Moreover, Jena provides extensive and comprehensive documentation of its persistence systems making straightforward to work with them. Finally, we also decided to implement upon Jena due to its wide–spread use.

We divide this section as follows: First, we give some details on ARQ and Jena, its storage model. We then provide a high level description of our approach. Next, we show how a selected query is made persistent using a data dictionary for saving space. Finally, we describe three ways in which ARQ can integrate such materialized results in query processing. Those will be evaluated separately in the next sections.

ARQ and the Jena Persistent Storage Schema

For our integration with ARQ we use the Jena persistence subsystem. This subsystem implements the Jena Model interface using a back-end relational database engine. The default Jena database layout uses a denormalized schema centered around a statement table, which essentially stores every RDF tripel as tuple. However, the values in the triple can either be included as value, or they are stored in other tables. Specifically, *short* literals are stored directly in the statement table, while *long* literals are stored in a literal table. Similarly, short URIs are stored in the statement table and long URIs are stored in a resources table. Table 2.8 and Table 2.9 describe the layout for those tables. Though this scheme helps to reduce space requirements especially in the presence of long and frequently used URIs or labels, it makes query processing more complicated as, for each row in the statement table, one must decide at runtime whether the respective value can be obtained directly or if a a join to another table is necessary.

Additionally, Jena defines system tables to store meta data. For further information we refer the reader to [92].

Table 2.8: Jena statement table for asserted (non-reified) statements.

Column	Type	Description
Subj	Varchar not null	Subject of asserted statement (ID or value)
Prop	Varchar not null	Predicate of asserted statement (ID or value)
Obj	Varchar not null	Object of asserted statement (ID or value)
GraphId	Integer	Identifier of graph (model) that contains the asserted statement

Table 2.9: Jena long literals table storing literals that are considered as too long to directly be stored in the statement table.

Column	Type	Description
Id	Integer not null	Identifier of long literal, referenced from the statement tables
Head	Varchar not null	First n characters of long literal (encoded)
ChkSum	Integer	Checksum of tail of long literal
Tail	Blob	Remainder of long literal (long literal without the head)

2.4.1 Implementation Overview

We differentiate two phases when working with materialized views as indexes. At offline-time, a predefined set of indexes is provided, analyzed, and their results are materialized. At query-time, queries are answered with the help of previous materialized results. We

divide the description of our implementation according to these phases[3].

Index creation. Indexes are created offline. Upon creation of an index, the following things happen. First, a new table is created, which will store the materialized query. The schema of this table is specific to the index: Each variable contained in the index pattern is represented as a field. Next, the query is executed, which leads to bindings for those variables. These are stored in the respective fields. At the end, every tuple in that table represents one result to the materialized query. During this process, we also create a data dictionary, which relates resource to unique identifiers. We only store those IDs in the index tables; this scheme is similar to the one used in Jena (see above Section 2.4), but we omit the costly choice between included and external values. These steps are executed only once per index (recall that index updates are beyond the scope of this work).

Index usage. At query-time, queries are analyzed and answered, possibly by using one or more of the indexes. This breaks down into the following steps:

1. Analysis of the query to find all maximal covers
2. Selection of the most suitable cover to answer the query given our cost model
3. Rewriting of the query using the chosen cover
4. Extension of the results of the cover to results of the query

Step one was discussed in Section 2.3 and step two will be addressed later in Chapter 3. Here, we concentrate on the third step, the query rewriting. Query rewriting can be performed in three different ways: i) using only ARQ, ii) by translation into SQL and access to the Jena native storage tables, and iii) by using a combination of ARQ and SQL. These different options will be discussed in Section 2.4.2.

Index Processing

Each index is materialized as a proper table in the underlying relational database. Its schema is formed by the set of different variables contained in the underlying query, regardless of whether the variables are contained in the SELECT clause of the query or not. Occurrences of the index in the dataset are stored as values for these fields. Each attribute of one tuple represents a binding for the respective variable. An example is shown in Listing 9.

```
CREATE TABLE Index1 (
    place          varchar(250),
    place_type     varchar(250),
    place_name     varchar(250)
);
```

Listing 9: Materialization of the query from Listing 40 as RDFMatView

[3]Note that, as mentioned in Chapter 1, the problem of selecting a set of materialized views from a given workload will be addressed later in Chapter 4 whereas here we specifically describe a methodology to select a suitable set of predefined indexes to cover a given query.

During the creation of an index we also calculate and store some properties, for instance size and frequency, which are used later to assess query execution plans. This information is stored in a single metadata table and loaded into memory once a query is executed. Notice that the memory usage of the statistics is very limited compared with the size of the dataset. Actually, the amount of memory of our statistics is in the order of *O(n)* where n is the number of predefined queries for the dataset.

2.4.2 Executing a Query Using RDFMatView

Query processing using materialized views usually combines results of multiple indexes. However, it is not always possible to cover all patterns of the query. The set of uncovered patterns is referred to as *residual part of a query.*

Definition 2.17 (Residual part of a query) *Let Q be a SPARQL query and C a random cover of Q where $|P(Q)| < |P(C)|$. r is defined as the residual part of Q iff $P(r) \subset P(Q)$ and $P(r)$ cannot be covered by C.* □

To completely answer a query, it is necessary to combine the results of the selected indexes with the results for the residual part of the query.

In the RDFMatView implementation, the covered pattern and the residual pattern are executed independently on the database and their results are joined either in memory or using SQL and temporary tables. There are, however, different ways to execute the residual pattern.

Example 2.18 (Processing residual patterns) *Let q be a query pattern and let i_1 and i_2 be two eligible indexes for q where:*

`q={(?a,p1,?b),(a?,p2,?c), (a?,p3,?d), (a?,p4,?e)}`

i_1`={(?x,p1,?y),(?x,p2,?z)} and,`

i_2`={(?x,p2,?y),(?x,p3,?z)}.`

Clearly, patterns $\{(?a, p1, ?b), (a?, p2, ?c), (a?, p3, ?d)\}$ can be processed by using the partial results of i_1 and i_2. Note that this process implies no access for the source RDF dataset, i.e. no subgraph comparison is performed. However, neither i_1 nor i_2 are able to cover the residual query pattern $(a?, p4, ?e)$ and therefore, the set of solutions for q is not complete.

Getting the complete set of solutions of q would require to query the residual pattern $(a?, p4, ?e)$ over the RDF dataset and join the results with the partial results of i_1 and i_2. Nevertheless, the complexity of the residual pattern of q is remarkably lower than the original pattern of q.

We studied three different methods to perform this task. First, the residual part is answered using the ARQ execution engine and has to be materialized on the client side. The covered pattern is answered from the materialized results of the index patterns and finally joined with the residuals from ARQ.

In the second strategy, instead of using ARQ to process the residual part of the query, we rewrite it to SQL using the native Jena database layout. The execution of the resulting SQL query is performed direct by the database query engine using stored procedures. The principal advantage of this method is that the join is completely done by the database server, however, its main drawback lays on its strong dependence with the database system.

The third strategy combines both previous methods and uses ARQ to process the residual part of the query, as well as the database server to join both partial results.

These strategies are explained in detail in the next sections.

2.4.3 Method 1: MatView-and-ARQ Engine

MatView-and-ARQ is a rewriting engine built on top of the Jena Framework. Given a query and a cover, it computes the set of residual patterns of the query and uses ARQ to execute this (sub-)query. Furthermore, it computes the result of the cover by joining the respective tables according to the variable mappings of the embeddings forming the cover. Results are also joined with the data dictionary to obtain RDF values, and finally joined to the result of the ARQ query to produce the complete answer to the original query. This engine encapsulates the logic for the execution of the cover and provides total independence from the underlying relational database system. Partial results from covered and uncovered patterns are joined in memory on the client side using a Sort–Merge Join (also known as Merge–Join) algorithm. Figure 2.7 illustrates the workflow of this engine.

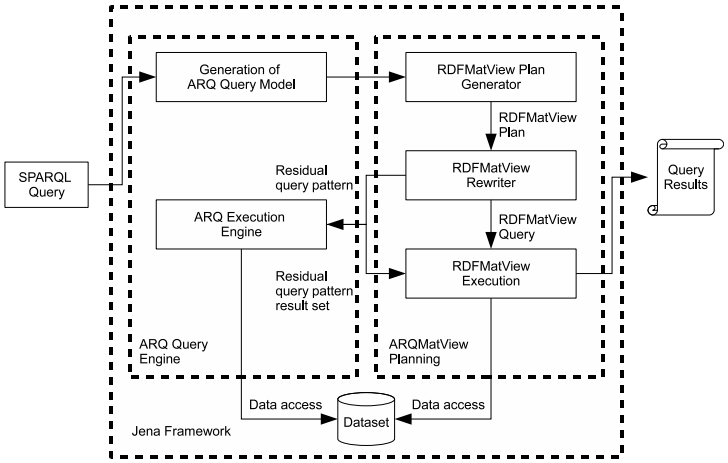


Figure 2.7: Workflow of the query processing using Matview-and-ARQ.

2.4.4 Method 2: MatView-to-SQL Engine

MatView-to-SQL is a rewriting engine, which unlike our first method, translates the residual part of the query into a SQL query on the Jena tables using an algorithm proposed by Chebotko in [24]. The SQL query is executed by the RDBMS. The result set is processed

using our RDF Dictionary and finally combined with the results of the cover. The complete query processing is performed inside the database execution engine using a stored procedure. Figure 2.8 illustrates the workflow of this engine.

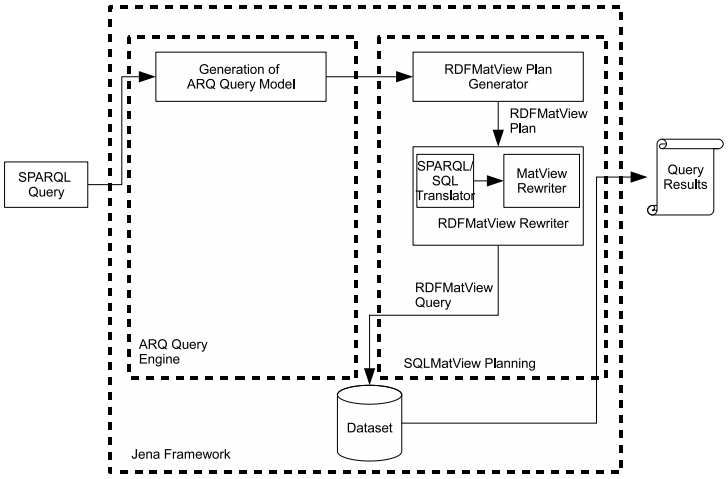


Figure 2.8: Workflow of the query processing using MatView-to-SQL

2.4.5 Method 3: Hybrid Engine

The third method is a mixture of MatView-and-ARQ and MatView-to-SQL. As in Method 1, after rewriting the query, this engine transfers the residual patterns to the query execution engine of ARQ. The second part of the process joins the results of the residual patterns with the resulting set of the covered part of the query patterns at server side. However, contrary to Method 1, this engine is database-dependent since this task is performed inside the database execution engine, as in Method 2. Figure 2.9 illustrates the workflow of this engine.

2.5 Evaluation and Results

In this section, we describe the evaluation of our approach. In essence, we concentrate on selecting which of our rewriting methods is the most efficient in terms of query processing disregarding any cost estimation[4]. We analyze performance and behavior of each method by manually selecting three covers (generated using the algorithms described in Section 2.3) for each query based on the number of participating indexes and residual patterns.

We use two widely-accepted SPARQL benchmarks: the Berlin SPARQL Benchmark (BSBM) [10] and the SPARQL Performance Benchmark (SP^2B) [80]. The domain of

[4]In Chapter 3 we describe different cost models to estimate an optimal or close to the optimal cover for a given query

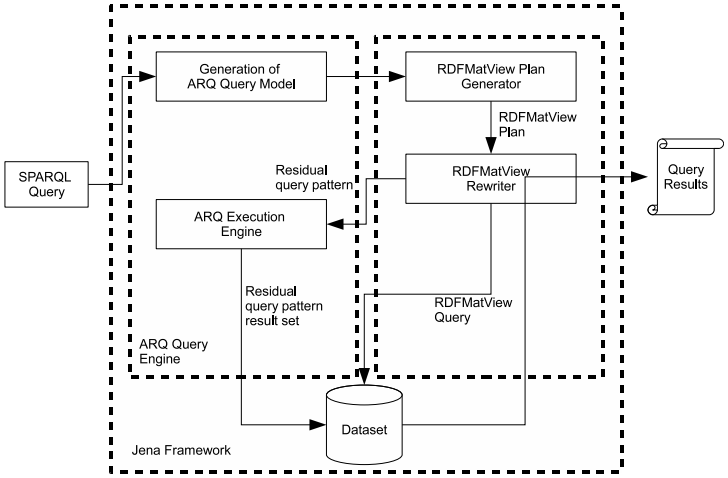


Figure 2.9: Workflow of the query processing using our hybrid engine.

BSBM is settled on e-commerce use case information whereas SP^{2}B benchmark is regarding bibliographic information about the field of Computer Science and, particularly, databases (DBLP [58]). Using data generators provided in these benchmarks we create the datasets required to perform our experiments.

As SPARQL framework, we use the ARQ/Jena RDF Storage System (version 2.5.7) on Postgres 8.2.

We generated eight RDF datasets with sizes ranging from 250K to 10M triples and tested the impact of the indexes on six different queries (three queries for each benchmark). A detailed description of queries and indexes can be found on Appendix A and Appendix B. For each query, we manually defined a set of indexes, leading to covers composed of one to three indexes. Our intention here is not to find the best set of indexes given a workload (index selection); instead, we study to which degree indexes that use different processing schemes speed up the execution. Our results describe the time required to process a query using the selected covers. Each cover is executed seven times, maximal and minimal values are excluded and average results are presented.

In the following, we first briefly introduce both benchmarks (BSBM and SP^{2}B). We then describe the datasets and test queries as well as the indexes we used. Finally, we present and discuss the results of our evaluation.

Berlin SPARQL Benchmark

The Berlin SPARQL benchmark is built on an e-commerce use case in which a set of products is offered by different vendors and where consumers have posted reviews about products [10]. The main classes of its schema are:

- *Product* Captures products with different sets of properties and features.

- *ProductType* Classifies products into a hierarchy.
- *ProductFeature* Represents product features for a specific product depending on the product type. Each product type in the hierarchy has a set of associated product features, which leads to some features being very generic and others being more specific.
- *Producer* Represents the producer of products.
- *Vendor* Represents the supplier of products.
- *Offer* Describes an offer to a product.
- *Person* Captures all person-related information.
- *Review* Provides ratings of a product.

The benchmark provides a data generator which supports the creation of arbitrarily large datasets using the number of products as scale factor. Table 2.10 provides a detailed description of the datasets using three different scale factors.

Table 2.10: Berlin SPARQL benchmark. Scaling and dataset population. The number of products is used as scale factor.

Number of Products	**666**	**2,785**	**70,812**
Number of RDF Triples	250,000	1,000,000	25,000,000
Number of Producers	14	60	1,422
Number of Product Features	2,860	4,745	23,833
Number of Product Types	55	151	731
Number of Vendors	8	34	722
Number of Offers	13,320	55,700	1,416,240
Number of Reviewers	339	1,432	36,249
Number of Reviews	6,660	27,850	708,120
Number of Instances	23,922	92,757	2,258,129
File size Turtle (unzipped)	22 Mb	86 Mb	2.1 Gb

The benchmark also defines a set of SPARQL queries to simulate a use-case driven workload. This set emulates the search and navigation pattern of a consumer looking for a product. Listing 10 shows a BSBM query example that returns a set of products satisfying a specific feature. For a detailed description see Appendix A Section 1.

SPARQL Performance Benchmark (SP^{2}B)

SP^{2}B is a language-specific SPARQL performance benchmark built upon the DBLP scenario, which comprises both a data generator and a workload of SPARQL queries. The

```
SELECT * WHERE {
    ?product a bsbm:ProductType39.
    ?product rdfs:label ?label .
    ?product bsbm:productFeature bsbm:ProductFeature2035> .
    ?product bsbm:productPropertyNumeric1 ?value1 .
}
```

Listing 10: Returns products with a given feature

generated documents mirror key characteristics and distributions encountered in the original DBLP dataset, while the queries implement meaningful requests on top of this data, covering a variety of SPARQL operator constellations and RDF access patterns [80]. Figure 2.10 shows the RDF representation of a DBLP instance.

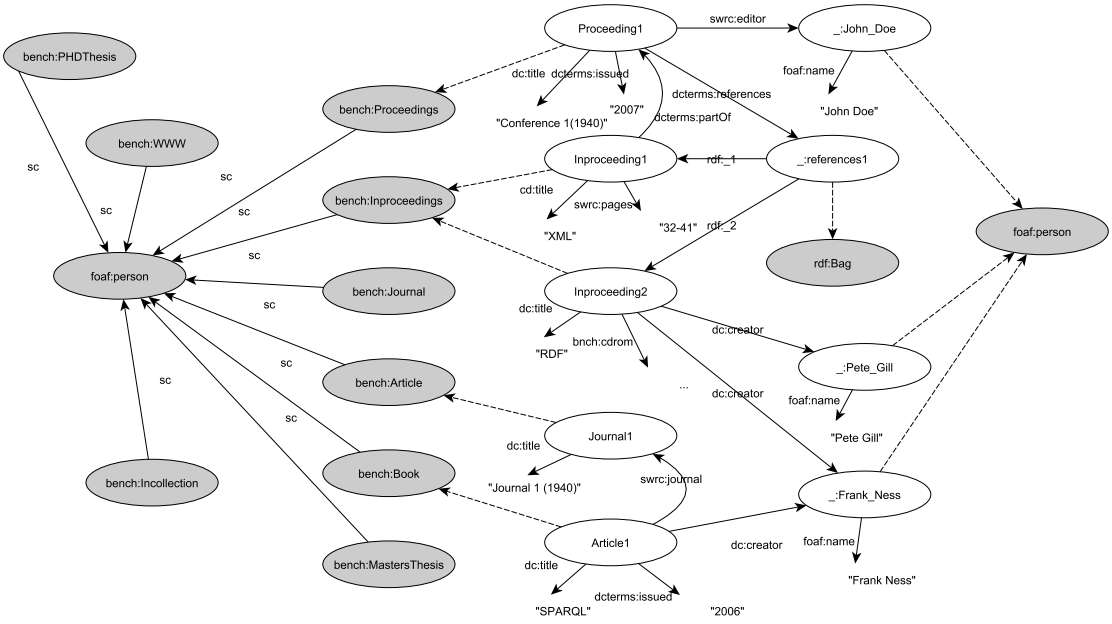


Figure 2.10: RDF representation of a DBLP instance as used in SP^{2}B dataset.

The use of DBLP documents in RDF format in SP^{2}B provides the benefits of creating on-demand large documents with data that contains real-world characteristics, i.e. natural correlations between entities, such as power law distributions (found in the citation system or the distribution of papers among authors) and limited growth curves (e.g., the increasing number of venues and publications over time). These facts are encapsulated in the data generator, which relies on a depth study of DBLP by comprising the analysis of entities (e.g. articles and authors), their properties, frequency, and also their interaction. On the

logical level, schema and instance layer are distinguished using gray and white elements respectively. For simplicity, dashed lines represent edges labelled with *rdf:type* and label *sc* in schema level edges represents an abbreviation for *rdfs:subClassOf*.

Similar to BSBM, this benchmark provides a data generator, which supports the creation of arbitrarily large datasets. However, this time the number of triples or the size of the output file are used as scale factors. Additionally, SP^2B defines a set of SPARQL queries to emulate the search patterns of a user looking for bibliography. In the following we show two example queries (for a complete description, see Appendix B Section 3).

```
SELECT ?yr WHERE {
?journal rdf:type bench:Journal .
?journal dc:title "Journal 1 (1940)"^^xsd:string .
?journal dcterms:issued ?yr
}
```

Listing 11: Return the year of publication of 'Journal 1 (1940)'

```
SELECT ?article WHERE {
?article rdf:type bench:Article .
?article swrc:pages ?value
}
```

Listing 12: Select all articles with property 'swrc:pages'

2.5.1 Datasest and Queries

For each benchmark, we create four datasets containing 250K, 500K, 1M and 10M triples, respectively. As these datasets have identical value distributions but different sizes, our evaluation concentrates on the scalability of our methods in different domains and we compare our three rewriting methods for RDFMatView looking towards selecting the most efficient. Based on the number of triple patterns we chose three queries for each benchmark. We transformed the query patterns into simple graph patterns and removed most bindings to variables. Bounded variables incur high selectivity resulting in the retrieval of only a handful of triples. Such queries are well supported by existing index structures and do not require the type of join-optimization that is achieved with our optimization technique. Therefore, performance gains would be only marginal. Our test queries are described in Listing 13.

From the queries described in Listings 13, we derive two sets of indexes containing 12 and 8 indexes respectively. Each index covers two to six patterns from at least one query. However, none of them completely covers a query. We concentrate on evaluating covers containing either a combination of indexes and possible a residual part of the query since most real-life SPARQL queries comply with this case.

```
Query1: Finds products for a given set of generic features.

Query2: Retrieve basic information about products.

Query3: Retrieve in-depth information about products including offers and
    reviews.

Query4: Extract all information about inproceedings documents.

Query5: Select all pairs of articles of an author that have been published
    in the same journal.

Query6: Return for each year, the set of all publications including the name
     of the authors.
```

Listing 13: Test queries derived from BSBM and SP^{2}B

2.5.2 Results

For each benchmark we evaluated three queries over four datasets using our three RDFMatView methods and plain ARQ (without indexes). We refer to the approaches to query execution as M1 for MatView-and-ARQ, M2 for MatView-to-SQL, M3 for the hybrid approach, and ARQ for plain ARQ. Our experiments use three different covers for each query and evaluate their processing time for the same query. As mentioned in Section 2.5, we use the algorithms provided in Section 2.3 to generate covers. To evaluate query processing time, we manually select three covers regarding the number of participating indexes and residual patterns. Table 2.11 shows the covers selected for each query. A complete list of covers can be found in Appendix A Section 2 and Appendix B Section 4.

Table 2.11: Selected covers for the evaluation of test queries using BSBM and SP^{2}B. The numbers identify the index of each cover as given in Appendix A and Appendix B

Query	$Cover_1$	$Cover_2$	$Cover_3$
$query_1$	1	4	8
$query_2$	1	5	6
$query_3$	1	6	12
$query_4$	1	8	10
$query_5$	1	8	17
$query_6$	1	3	5

All queries were executed seven times using each cover, which amounts a total of 21 times per query on each dataset. Maximal and minimal values are excluded and a set of statistical measures are reported such as mean, median, standard deviation, max and min.

Figures 2.11 to 2.14 show individual processing for $query_1$ over BSBM and $query_4$ over SP^{2}B using four datasets ranging from 250K to 10M triples. Further results can be found

on Appendix C Section 5. Rows denote query processing time over each dataset using our three rewriting methods, i.e., M1: MatView-and-ARQ; M2: MatView-to-SQL; M3: Hybrid, and ARQ: plain ARQ (time in milliseconds).

Upon analysis of these results, M1 and M3 are more effective than M2 independent from the extension and domain of the datasets. Specially in BSBM datasets, *query*2 and *query*3 demand larger processing than the standard engine when queries are executed with method M2. When using method M2, the Jena native storage schema is directly accessed during processing of the residual part of the query. Since the values are encoded following the Jena layout, our process needs to parse the stored values, extract the required information and perform joins between related tables, which increases the processing time. An improvement for this method could be to use a different triple layout that simplifies the process of querying the data using a translated SPARQL query. However, it would imply to modify the rewriting process of the query since it depends on the underlying database schema.

Results also evidence that selection of an optimal cover is a decisive factor for improving or worsening query execution time. Recall that our evaluation concentrates on finding which of our methods is the most efficient in terms of processing and not on providing the best way to process a given query. This task is addressed further in Chapter 3. There, we provide two different models to evaluate and estimate values for each possible cover of a query regarding different statistical measures about the resources that are consumed (query, indexes and RDF data).

Figure 2.15 and Figure 2.16 summarize the results shown from Figure 2.11 to Figure 2.14 respectively. As mentioned for Figure 2.11, methods M1 and M3 are more efficient than M2 at execution time regarding the analyzed covers for the same queries in all datasets.

Figure 2.15(a) and Figure 2.16(a) show that minimal and maximal processing time can remarkably vary depending on the selected cover used to execute the query. These differences show that finding the "right" cover to execute the query is an important task to improve SPARQL query processing.

We select M1 as the most promising rewriting method from our three options. Clearly, it provides efficient processing time and is also independent from the database schema, which is not the case when using method M3. Therefore, in the rest of this thesis we perform our experiments using RDFMatView with rewriting method M1.

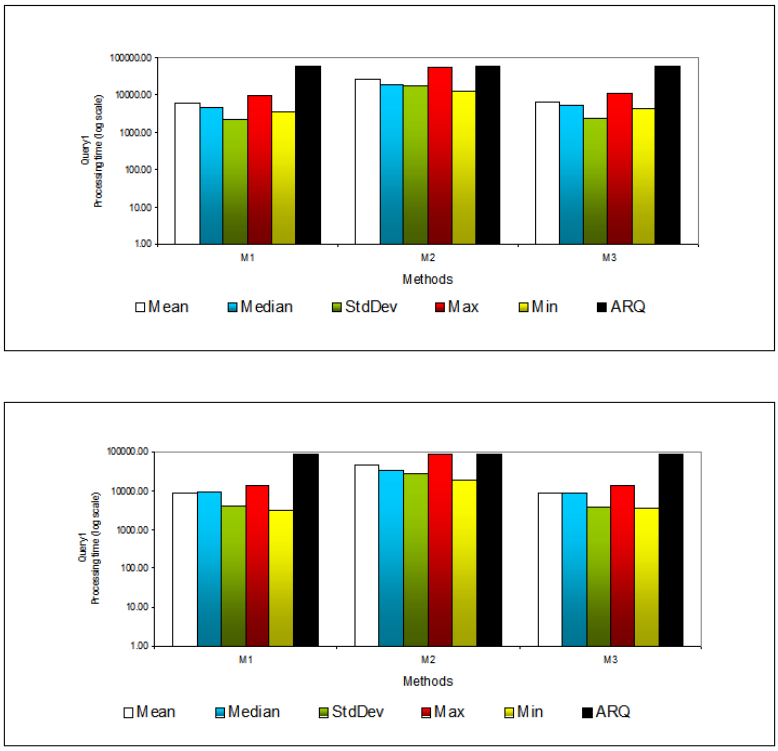


Figure 2.11: Processing of query1 over BSBM using 250K and 500K triples datasets. Results show that rewriting methods M1 and M3 are more efficient than M2.

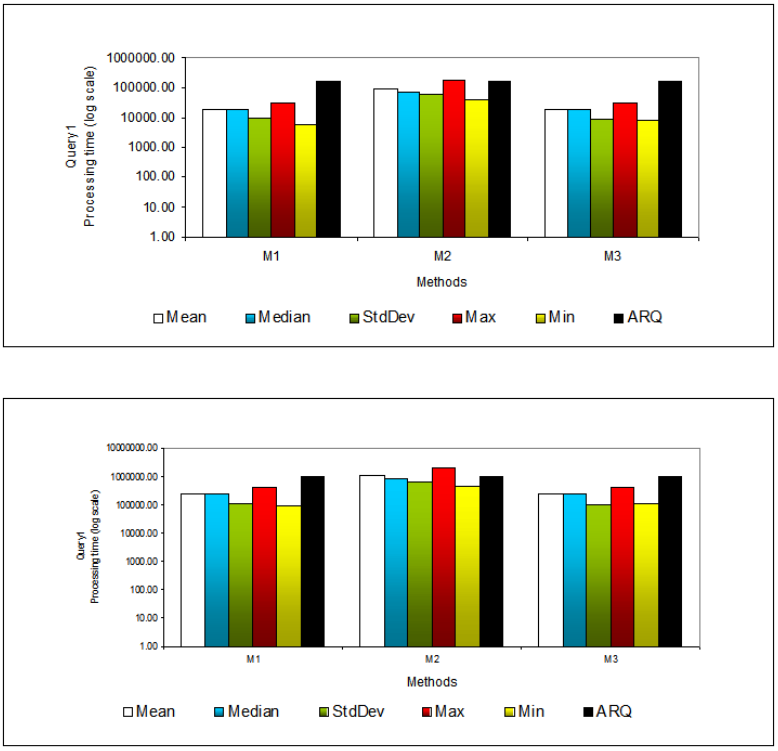


Figure 2.12: Processing of query1 over BSBM using 1M and 10M triples datasets . For explanation see Figure 2.11.

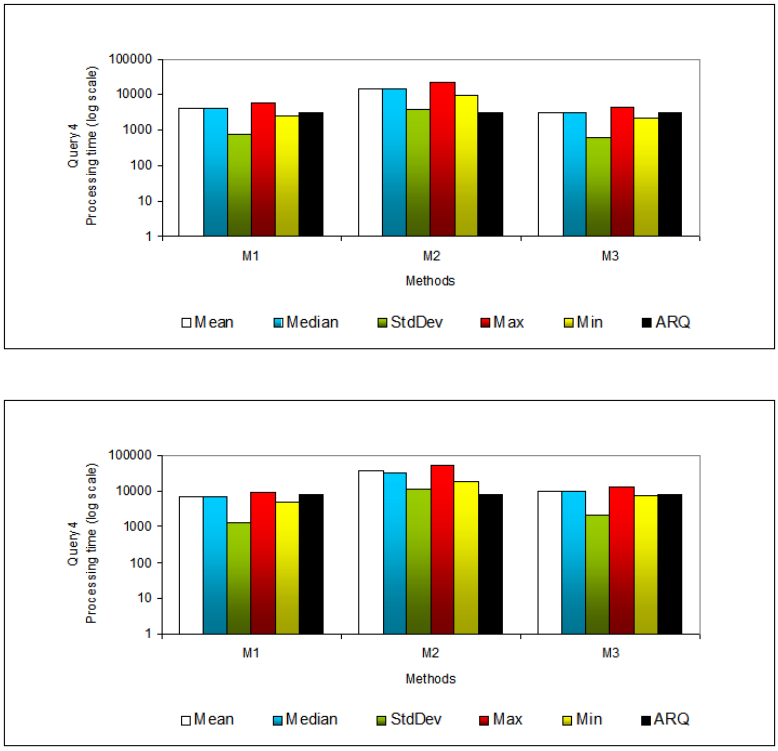


Figure 2.13: Processing of query4 using rewriting methods over SP^{2}B using 250K and 500K triples datasets. Results show that all rewriting methods are capable to improve standard processing time, depending on the cover selected to execute the query.

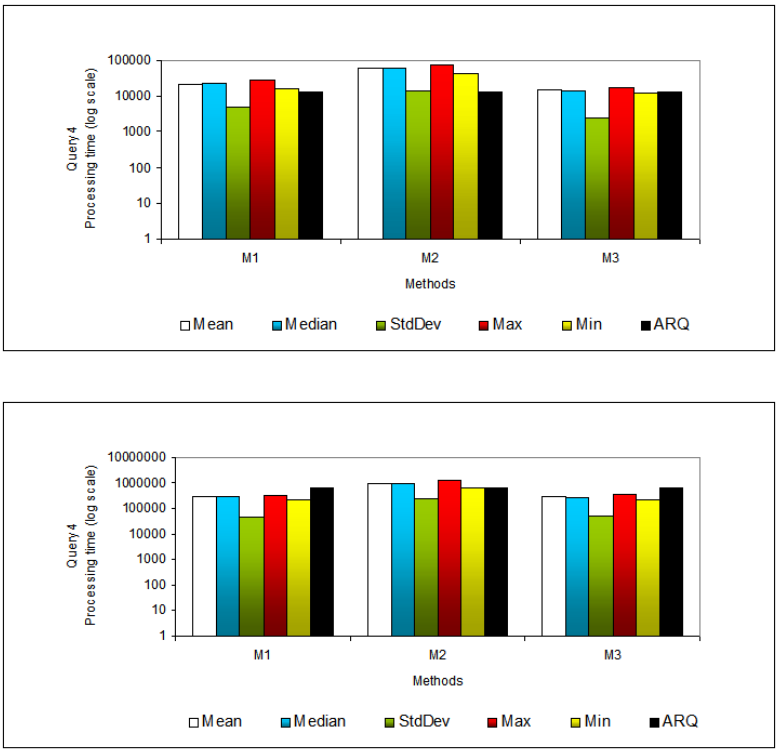
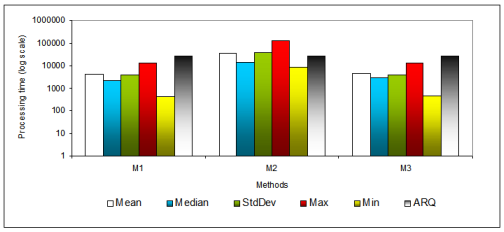
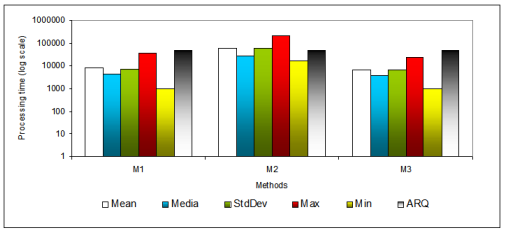


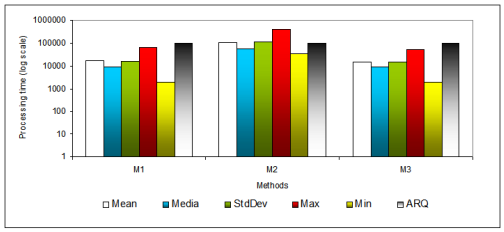
Figure 2.14: Processing of query4 using rewriting methods over SP^{2}B using 1M and 10M triples datasets. For explanation see Figure 2.13.



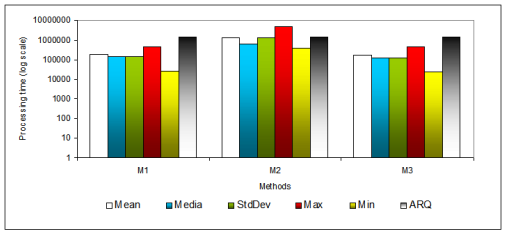
(a) Statistics BSBM 250K triples



(b) Statistics BSBM 500K triples

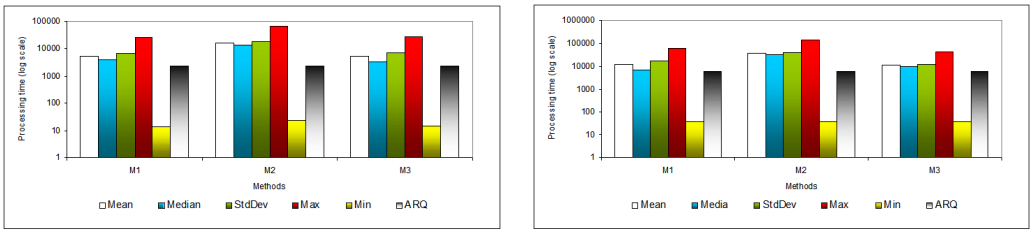


(c) Statistics BSBM 1M triples



(d) Statistics BSBM 10M triples

Figure 2.15: Contrary to Figure 2.13 where results depict processing for each query, here we evaluate the complete workload and compare the methods using BSBM with 250K, 500K 1M and 10M triples. Again, M1 and M3 evidence better performance than M2. Nevertheless, execution time noticeably varies according to the cover used in the processing of the query. Note that y-axis is plotted in log-scale.



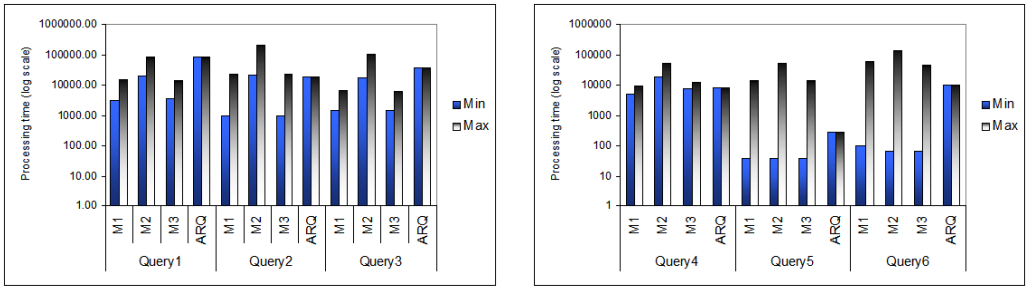
(a) Statistics SP^{2}B 250K triples

(b) Statistics SP^{2}B 500K triples

(c) Statistics SP^{2}B 1M triples

(d) Statistics SP^{2}B 10M triples

Figure 2.16: Processing time for test queries using SP^{2}B with 250K, 500K 1M and 10M triples. For explanation see Figure 2.15.



(a) Maximal and Minimal BSBM

(b) Maximal and Minimal SP^{2}B

Figure 2.17: Maximal and minimal processing time using BSBM and SP^{2}B 500K triples datasets. The range between maximal and minimal values noticeably varies with regards to the selected cover and the residual part of the query. Evidently, selecting covers that optimize this time is an important task to improve SPARQL query processing.

Limitations

Our current approach using materialized views as indexes for SPARQL queries has a number of limitations:

- The *WHERE-CLAUSE* of a SPARQL query must be a *Basic Graph Pattern*.
- Filters and modifiers are not considered.
- Blank nodes are not allowed.

In future work, we plan to extend our implementation, especially to allow the use of filters and modifiers.

2.6 Summary and Related Work

In this chapter we formally introduced RDF and SPARQL concepts. We also set the basis for the design of RDFMatView, a novel indexing approach for RDF, based on materialized SPARQL queries. The objective is to select a set of index patterns and materialize its occurrences to answer subsequent queries more efficiently.

The use of RDFMatView aims to reduce the number and the extension of isomorphism tests, which finally entails a reduction of the query processing time. This is the case whenever a subset of triples of the query patterns is covered using one or more RDFMatView patterns. This covered part of the query is pre-computed and persistently stored allowing to retrieve its results without searching directly in the data graph. Additionally, we also introduced the concept of overlapping indexes as an important property, which allows to select an optimal set of indices (covering as many query patterns as possible) to improve query processing. Furthermore, we provide algorithms that encapsulate the functionality required for answering SPARQL queries using materialized queries. To show that our ideas are viable for RDF and SPARQL, we developed three rewriting methods and used a standard SPARQL engine to process queries and compare processing time. Results show that remarkable savings in time can be achieved when using our methods. There are, however, cases in which, for the same query, processing time increases according to the chosen set of indexes at execution time. These cases should be avoided by suggesting an "optimal" execution plan regarding a statistical cost model. We address this topic in Chapter 3.

Related Work

Some approaches for SPARQL query processing have proved to be very efficient by providing either a relational or native RDF data schema. For instance, in [1] Abadi et al. propose a vertical partition approach for Semantic Web data management. An enhancement of this approach is proposed by Weiss et al. in [91]. Therein, RDF data is indexed in six possible ways, i.e., an index for each possible ordering of the three RDF elements. Each instance of an RDF element is associated with two vectors; each such vector gathers

elements of one of the other types, along with lists of the third-type resources attached to each vector element. This scheme is capable of speeding up single joins tremendously, but storage requirements are very high, which becomes a serious issue when using huge datasets.

Neumann and Weikum developed *RDF-3X*, a SPARQL engine resembling a RISC-style architecture with specifically-designed data structures and operations [68]. The authors overcome the "giant-triples-table" [1] bottleneck by creating a set of indexes and a fast way for processing merge joins. Similar to [91], RDF-3X maintains six possible permutations of subject, predicate and object in six separate indexes. The authors also present a compression algorithm to decrease the space consumption.

All these approaches have in common that they dedicate their efforts on indexing the relational representation of the RDF data. When faced with queries consisting of multiple basic graph patterns, they still have to compute multiple joins (although every single join is faster). In contrast, our work specifically targets the speed-up of complex queries consisting of many basic graph patterns by indexing complete query patterns.

There is some other work along this line. In [89] the authors present *GRIN*, a lightweight indexing mechanism for RDF data. The idea is to draw circles around selected *center* vertices in the graph where the circle would comprise those vertices in the graph that are within a given distance of the "center" vertex. Basically, GRIN is a binary tree where the set of leaf nodes form a partition of the set of triples in the RDF graph. An interior node represents the set of all vertices in the RDF graph that are within a specific distance. To evaluate a query, GRIN derives a set of inequality constraints from the query. These constraints are evaluated against the nodes of the GRIN index.

A similar indexing approach is presented in [37]. This work proposes a set of indexes of precomputed joins created from all possible join combinations between triple patterns. As [89], this approach creates a general purpose set of indexes based on joined triple patterns, but the number of indexes to manage is impractical when the number of joined triples is $>= 3$.

The two systems just described index larger portions of the RDF dataset and not just single triples. However, they propose to apply their techniques to all RDF triples, while we only build user-chosen indexes. Our work fundamentally is based on the assumption that some patterns are combined more frequently than others, and that only indexing those combinations promises to provide large speed-ups at manageable space and maintenance cost.

The differences between our ideas and that of other RDF indexing schemes can be described by drawing a parallel to B*-indexes in relational databases [26]. Nobody would suggest to speed up queries by indexing every attribute; instead, systems assume that developers have a rough idea about the types of queries that need to be answered and therefore index only the relevant attributes. Furthermore, optimal speed-up can only be achieved when also combinations of attributes can be indexed, and not only single attributes. In this sense, the former approaches index every single attribute, the latter indexes every possible combination of attributes, and we suggest to index only selected combinations of attributes.

3 Cost Models

Cost models play a very important role in query optimization [31]. Basically, query optimization explores different alternatives to execute a query and chooses one of them as the best candidate for later execution. This objective is achieved by considering three major aspects:

- Generation of execution plans
- Search strategy
- Cost models

The first and second aspects are in charge of generating a set of logically equivalent plans to execute a query. In general, there are three steps that are considered by the query processor: i) query parsing, ii) generation of logical plans and, ii)generation of physical plans. Query parsing describes the process of translating a query given in a SQL–like language into an expression tree. From these expressions, the tree is transformed into a set of algebraic operations that forms a logical plan. Note that from a parse tree several logical plans can be generated depending on how the operations are associated. Physical plans describes specifically how the set of operations is applied, algorithms used on each step, as well as how the stored data is managed among operations [33]. Since a given query can potentially be evaluated by a large number of different plans, those plans are enumerated by using a specific search strategy (Lanzelotte et al. in [55]). The third aspect determines from the set of enumerated plans, which alternative is the best to answer the query in terms of execution time. This is usually achieved by using a cost function that estimates and assigns a value to each plan.

In Chapter 2 we introduced RDFMatView, an approach to optimize SPARQL query processing. RDFMatView optimizes query processing according to these aspects. The first step is addressed by generating execution plans using a predefined set of indexes (see Definition 2.9). Those plans are referred here as *covers* and consist of sets of indexes with overlapping relations (See Definition 2.12). We faced the second step by restricting our search space to contain only maximal overlapping covers (see Chapter 2). This condition avoids an explosion of the number possible covers for a given query. In this chapter, we turn our attention to the third step, i.e., cost models. A cost model is used as a basis for comparing different plans and for selecting the best plan (regarding our model) for query execution. Basically, the cost model estimates the execution time of each cover regarding information gathered from the resources contained in the query pattern and the dataset.

This chapter is structured as follows. In Section 3.1 we describe general approaches used in relational database systems for estimating costs. We then analyze three different

cost models to evaluate execution plans (covers) over RDF data. We describe in Section 3.2 a cost model based on the selectivity of a cover proposed by Heese et al. in [48]. Using this model, we estimate the possible number of occurrences a cover may have by combining selectivity of each participating index in the cover. Section 3.3 describes a second model, proposed by Moebius in [65], based on estimated cardinalities for query patterns. Whereas the first model estimates costs regarding the covered query pattern, the second model also takes uncovered patterns into account. In Section 3.4 we introduce a novel cost model based on join size estimation between indexes and uncovered patterns. With this combination of features we strive to add more accuracy to the cost estimation. An evaluation of these three models is provided in Section 3.5 and we conclude this chapter with a discussion and related work in Section 3.6.

3.1 Cost Models in Relational Database Systems

Query processing and optimization has always been a critical component of database technology. Both deal with efficient and effective processing of user queries against a database. We can formulate their main goal as follows: Finding the data from a usually large database that matches a certain query as efficiently as possible [95].

To achieve this goal the system requires estimations on the cost of each possible execution plan for a given query. This cost estimation makes only sense if it roughly correlates with the real processing time. Evidently, we cannot know exactly the cost of each plan without executing it. Therefore, we need to estimate the cost of each plan using suitable assumptions and statistical measures [33].

In [52] Ioannidis defines cost models as specific arithmetic formulas that are used to estimate the cost of execution plans. These formulas may contain different sub–formulas that represent estimations for different components in an execution plan e.g., join method, table access methods or used indexes. Formulas are given mostly as simple approximations of what the system does during query processing. These approximations are usually based on assumptions concerning buffer management, disk-cpu overlap, I/O access type, distribution of values, size of tables, etc.

In large database systems the cost to access data from disk is usually the most important factor to consider. Evidently, disk access is by orders of magnitude slower compared to in–memory operations. Moreover, CPU speeds have increased faster than disk speeds [84]. To answer a query we usually require to manage intermediate results. Whenever these results exceed the size of the reserved buffers, they need to be stored. Storing these results requires a certain amount of disk access that is directly related with the size of the intermediate relation. Therefore, our research has been focused on describing the estimation of sizes of intermediate relations, i.e., the number of possible results for a given relational query.

Cost estimation does not strive to discover the exact cost (i.e., required time) for a given operation, but to provide enough information that help to select an “optimal” plan. In other words, the unit of the estimates may be purely abstract but the values should correlate with the processing time of the plan, i.e., the least cost should be assigned to

the best physical plan, and high costs to bad plans.

Estimating Parameters

By now, we have mentioned the basic factors considered to estimate query processing, where data access from disk has been point out as a large costs generating task. In this section, we describe how intermediary result set sizes are estimated in a relational database context and how the parameters required for these estimations can be obtained from a given database.

The necessary statistics usually are computed in advance. The basic parameters for a relation R and an attribute a are described as follows:

- The size of a relation R counts the total number of tuples in R. We denote it as $|R|$.
- The variety of a relation R counts the number of different values assigned to attribute a in R. We denote this parameter as $variety(R, a)$.

Both parameters can be obtained from the database by scanning R and counting the number of tuples and the number of different values assigned to every a. If a table is extremely large, these values may be estimated for instance by sampling [42]. It is possible to derive other parameters from these values, e.g. the number of blocks in which R can be stored.

In relational databases, there is a set of strategies that can be applied to estimate sizes of intermediary results regarding these parameters [33, 75, 84]:

- Estimating the size of a projection

  A projection over a relation produces results that may decrease or increase the output length of the tuples. The more attributes are requested, the larger is the size of the result. Note that the number of results is not reduced, but only the length of each contained tuple.

- Estimating the size of a selection based on a single attribute (equality comparison)

  The result size of a selection operation depends on the selection predicate. The basic estimate for the selection size over a given relation R is described by the ratio of the total number of tuples in R and the number of different values of the selection predicate in R.

- Estimating the size of a selection based on conjunctive conditions

  This case is a generalization of the previous estimation. In general, we cannot early estimate the size of this selection treating the multiple selection as a sequence of simple selections. Often, statistical independence can be assumed, in which case each simple selection evaluates only one condition and the size of the results equals the size of the original relation multiplied by the *selectivity*[1] factor for each condition.

[1] Here, selectivity describes the probability that a tuple in R satisfies a given selection condition

- Estimating join sizes between two relations based on a single attribute

  Estimating the size of a natural join is more complicated than estimating the size of a selection. As always we do not know exactly how two relational tables R and S relate by means of a given attribute a; yet we can estimate it. For instance, assume that we analyze joins between two relations involving only the equality of two attributes $R.a$ and $S.a$:

  1. R and S have disjoint sets of a–values. In this case the size of the join is 0.
  2. Attribute a is primary key of S and foreign key of R. In this case, each tuple of R exactly joins one tuple of S and therefore, the size of the join equals the size of R.
  3. All tuples of R and S share the same set of a–values. In this case, the Cartesian product between R and S is the most accurate estimation for the size of the join.

- Estimating join sizes between two relations based on multiple attributes

  This case is a generalization of the previous estimation. In this case, instead of a single attribute we have a set of attributes X common to R and S, where $|X| \geq 2$. Assuming statistical independence, the idea is to estimate the size based on the probability that a row r in R and a row s in S agree on each given attribute $x_i \in X$ where $i = 1...n$.

- Estimating join sizes among multiple relations

  This is the most general case for a join. Similar to the previous strategy, the size estimation is based on the product of single incidence probabilities among rows, present in a k number of relations by means of each join attribute.

Example 3.1 shows how the previously described parameters can be applied to estimate the size of a selection regarding a single attribute with equality comparison.

Example 3.1 (Using Estimation Parameters) *Let R be a relational table and a an attribute of R. Estimating the size of a selection over R using equality comparison using the size of R and the variety of R with respect to a, denoted by $|R|$ and $variety(R, a)$ can be expressed as follows:*

$$\frac{|R|}{variety(R, a)}$$

A strategy to gain more detailed knowledge on the data distribution in a given database is to build *histograms*. A histogram is a representation of the frequency of the values stored in a relational table [84]. The main goal of a histogram is to divide the values present for each attribute into ranges. According to each attribute–value, tuples are counted and associated to their respective range. Additionally, histograms usually store the number of distinct values contained in that range as well. In this sense, it is possible to know the frequency but also the variety of the attributes for each range.

There are different types of histograms used in database systems e.g. *equi–width* and *equi–depth* histograms. The former divides the range of values into equal sized ranges, whereas the latter adjusts the boundaries of the ranges such that each range has the same number of values [51].

While histograms are a well–established statistical asset in relational databases, they are difficult to apply in RDF data due to its usually heterogeneous and string–oriented nature [67]. Therefore, in the following we use the previously defined estimates, i.e., $|R|$ and $variety(R, a)$, as building blocks for our proposal of cost models adapted to the graph–nature of the RDF data model. We propose three different models that can be applied for SPARQL queries. Our goal is to evaluate different execution strategies for a given query using a set of materialized views as indexes. We seek to provide enough evidence to select the best option regarding our cost estimation.

3.2 SyCoM: A Selectivity Cost Model

In Chapter 2 we defined which sets of indexes are eligible for a given query. At runtime, the optimizer chooses between these options, or decides to execute the query without using indexes. This decision should be taken based on the expected savings with respect to execution time by using indexes in query execution. In the following, we present SyCoM, a simple model proposed by Heese et al. in [48]. We refine the model definition using the concept of embeddings (instance of an index) and integrate it into RDFMatView [18].

This model implicitly takes a number of assumptions on the data graph. For instance, all triples of a pattern are treated equally with respect to the expected numbers of results, disregarding the number of contained variables and the frequency of constants. These assumptions heavily simplify the model and allow us to estimate execution costs without any detailed knowledge of the underlying database but may lead to extremely bad estimations. A further problem is that SyCoM disregards patterns that are not covered by any index. Estimations are only computed based on those query patterns that can be replaced with results from a set of preprocessed indexes.

Preliminaries

SyCoM is based on the following fundamental observations. Recall that for each index I that occurs in the query pattern, each occurrence of the query pattern in the data graph must contain an occurrence of I.

1. It is beneficial to prefer indexes that have few occurrences. Every occurrence of an index must be validated to verify wether it can be extended to an occurrence of the query. Thus, the less occurrences an index has, the less time is required to answer the query.

2. It is reasonable to cover as many query patterns as possible. This process reduces the number of query patterns that need to be evaluated against the data graph. According to this, large index patterns (i.e., covers) are specially interesting.

We formally capture these observations in the definition of *selectivity* of an index. To calculate the selectivity we need the following information:

- The *size* of the index pattern. It represents the number of index patterns. We denote the size of I as $|I|$.
- The *frequency* of the index pattern. It represents the number of results of an index I over a dataset G. We denoted it as $\#(I)$.
- The *size* of the dataset. It represents the total number of triples in a dataset G. We denote this parameter as $|G|$.

Definition 3.2 (Selectivity of an index) *Let I be an index over a data graph G. The selectivity $s(I)$ of I is defined as the ratio of the number of occurrences of an index in a given graph over the possible total number of index occurrences in the graph:*

$$s(I) = \frac{\#(I)}{|G|^{|I|}}.$$

□

From the previous definition we derive our formula to estimate the selectivity of a set Υ of indexes. To this end, we view Υ as the union of the patterns of the indexes I in Υ (similar to the union of RDF graphs, see [70]). Without further knowledge, the selectivity of Υ is lower than the selectivity of all its indexes, because any occurrence of one index potentially can be combined with any occurrence of all other indexes. This leads to the following worst-case estimation for the selectivity of a set of indexes.

Lemma 3.3 (Selectivity of a set of indexes) *Let G be an RDF data graph and $\Upsilon = \{I_1, ..., I_n\}$ with $\Upsilon_i = (P_i, O_i), i = 1, ..., n$ be a set of indexes over G. Then, the following formula is an upper bound of the selectivity of Υ:*

$$sel(\Upsilon) = sel(I_1 \cap I_2 \cap \ldots \cap I_n) \leq \frac{\prod_{i=1}^{n} |O_i|}{|G|^{\max\{|P_1|,...,|P_n|\}}}$$

Proof: As any occurrence of one index in the worst case is combined with any occurrence of any other index, it follows that $O_\Upsilon \leq O_{I_1} \cdots O_{I_n}$. Further, the size of the index pattern of Υ is at least $|P_1 \sqcup ... \sqcup P_n| \geq \max\{|P_1|, ..., |P_n|\}$. Taken together, this gives:

$$sel(\Upsilon) = sel(I_1 \sqcup I_2 \sqcup \cdots \sqcup I_n) \leq \frac{\prod_{j=1}^{n} |O_i|}{|G|^{\{|P_1 \sqcup ... \sqcup P_n|\}}} \leq \frac{\prod_{j=1}^{n} |O_i|}{|G|^{\max\{|P_1|,...,|P_n|\}}}$$

□

Selectivity of Overlapping Embeddings

Lemma 3.3 assumes that we have no information about relationships between indexes of this set. However, we already defined several aspects on how indexes may overlap (see Chapter 2, Definition 2.11). Furthermore, we restricted query processing to use covers containing sets of overlapping embeddings (see Chapter 2, Definition 2.12), because the knowledge on overlaps among embeddings allows us to estimate the selectivity of a cover more accurately.

As explained in Section 2.2 intensional overlapping is only a necessary yet not sufficient condition for the existence of a concrete overlap in the underlying query. Actually, if two embeddings intensionally overlap in the query but do not extensionally overlap in the data graph, we can immediately conclude that the query has no answer. Therefore, there are two different cases which should be considered for the selectivity of a cover: i) The embeddings overlap intensionally but not necessarily extensionally and ii) the embeddings overlap intensionally and extensionally.

An embedding can be seen as an instance of its underlying index. Therefore, the selectivity of embeddings, that overlap intensionally but not necessarily extensionally, must be estimated in the same manner as the selectivity of a set of indexes (see Lemma 3.3). Thus, any occurrence of one embedding in the worst case can be combined with any occurrence of any other embedding.

Definition 3.4 (Selectivity of intensionally overlapping embeddings) *Let G be an RDF data graph, Q a query over G and $m = \{m_1, ..., m_n\}$ a cover[2] of Q. Then, we can estimate the selectivity of m as follows:*

$$sel_{Int}(m) = sel(m_1 \cap m_2 \cap \ldots \cap m_n) \leq \frac{\prod_{i=1}^{n} |m_i(O_i)|}{|G|^{\max\{|m_1|,...,|m_n|\}}}$$

where $m_i(O_i)$ means the projection of all occurrences of O_i using m_i. □

For the second case, when the cover consists of intensionally overlapping embeddings that also overlap extensionally, we can use a stronger estimation:

Definition 3.5 (Selectivity of extensionally overlapping embeddings) *Let G be a data graph, Q a query over G and $m = \{m_1, ..., m_n\}$ a cover of Q. Assume that all pairs of embeddings also mutually overlap extensionally. Then, we can estimate the selectivity of m as follows:*

$$sel_{Ext}(m_1, ..., m_n) \leq \frac{min(|m(O_1)|, ..., |m(O_n)|)}{|G|^{\max\{|m_1|,...,|m_n|\}}}$$

□

[2] With slight abuse of notation. By m we mean an instance of I by means of mappings between indexes and query patterns.

Note that, since all pairs of embeddings overlap extensionally (see Definition 2.12), the maximal number of selected occurrences is given by:

$$min(|m(O_1)|, ..., |m(O_n)|)$$

As stated in Chapter 2 and in compliance with Definition 2.12, we only consider intensionally overlapping embeddings in this thesis. Thus, in the following we estimate the selectivity of a cover using Definition 3.4.

Definition 3.6 (SyCoM: Selectivity Cost Model) *Let Q be a SPARQL query, C a cover of Q. Then, we estimate the cost of executing Q using C as*

$$c(Q, C) = sel_{Int}(C)$$

□

When selecting an optimal cover according to this model, those covers with lower selectivity values are preferred, i.e., those covers where the estimated relation between number of occurrences and possible total number of occurrences is smaller. This assumption captures that the lower the selectivity of a cover, the less time is required to combine occurrences of its participating indexes.

3.3 CardiOS:A Cardinality Cost Model for SPARQL

In the previous section (Section 3.2) we presented SyCoM, a cost model based on a specific form of selectivity that estimates which indexes are optimal for answering a given SPARQL query. However, even though our evaluation (see Section 3.5.5) shows that selecting plans using SyCoM achieves gainings in orders of magnitude compared to standard query processing, it also shows that it is far from being optimal and often fails to find the best or at least a good physical plan. In this section we describe a second model (CardiOS) proposed by Moebius in [65] that, given a predefined set of indexes and predicate–statistical metadata, evaluates all possible execution strategies and suggests an optimal plan regarding their estimated values. Compared to [65] we enhance this approach mainly in two parts. First, we analyze the treatment of cycles in the patterns, and second, we evaluate the model using different weights for the covered and residual patterns of the query. Our evaluation (see Section 3.5.3) shows that using predicate statistics to estimate the cardinality of queries patterns generates more accurate information than the selectivity to select an optimal execution plan.

This approach is structured as follows. In Section 3.3.1 we describe the *basis and the potential* as the two fundamental building blocks CardiOS is based upon. Sections 3.3.2 and 3.3.3 describe methodologies to compute both measures and Section 3.3.4 describes the treatment of cycles in query patterns. Section 3.3.5 describes how constants in patterns influence our estimation, and in Section 3.3.6 we combine these concepts and define the cost model CardiOS.

Preliminaries

In this section we introduce the necessary notation to explain this approach for estimating cardinality of query patterns based on definitions provided by Moebius in [65]. First, Definition 3.7 describes the set of subjects, predicates and objects in a set of triple patterns. We want to emphasize that a set of triples can be seen as an special set of variable–free triple patterns. Therefore, Definition 3.7 applies to both triples and triple pattern sets, i.e. datasets and query patterns.

Definition 3.7 (Subjects, Predicates, Objects) *Let P be a set of triple patterns or a dataset.*

- $\Sigma(P) := \{s|\exists(s,p,o) \in P\}$ *denotes the* **set of subjects** *in P.*
- $\Pi(P) := \{p|\exists(s,p,o) \in P\}$ *denotes the* **set of predicates** *in P.*
- $\Omega(P) := \{o|\exists(s,p,o) \in P\}$ *denotes the* **set of objects** *in P.*

□

Secondly, we introduce the term *element* [65]. An element of a triple pattern is any value that occurs at its first or third position. The concept of elements is derived directly from the graphical representation of RDF.

Definition 3.8 (Elements) *Let $T = (s,p,o)$ be a triple pattern and P a pattern.*

- *s and o are the* **elements** *of T.*
- *The* **set of elements** *in P is defined as:*

$$\Xi(P) := \Sigma(P) \cup \Omega(P)$$

□

Now we turn to define adjacent triples [65]. Basically, adjacent triples are those triple patterns that share an element.

Definition 3.9 (Adjacent triples) *Let P be a pattern and e an element contained in P. The set of* **adjacent triples** *to e is defined by:*

$$\Theta(P,e) := \{(e,p,o) \in P\} \cup \{(s,p,e) \in P\}$$

□

Looking at our definitions, it is evident that we treat predicates different than subjects and objects. Typically, most SPARQL queries are built using patterns with bound predicates [38]. Additionally, in RDF datasets, predicates describe relations between two

elements (subject and object). Thus, the number of different predicates is remarkably smaller than the number of different subjects and objects. By using this feature, we propose an estimation strategy regarding frequency of specific predicates. As we generate statistics over predicates, in the rest of this section, whenever we mention a query pattern we mean triple patterns that do not allow variables as predicates.

We exploit statistics on three measurements, namely frequency, subject cardinality and object cardinality. We compute these values for each predicate contained in a given RDF dataset. Definition 3.10 introduces these concepts formally [65].

Definition 3.10 (Frequency, Subject cardinality and Object cardinality) *Let D be an RDF dataset. For every predicate $p \in \Pi(D)$:*

- *the* **frequency of p** *is defined as:*

$$f(D,p) := |\{(s,p,o) \in D\}|$$

- *the* **subject cardinality of p** *is defined as:*

$$c_s(D,p) := |\{s|\exists(s,p,o) \in D\}|$$

- *the* **object cardinality of p** *is defined as:*

$$c_o(D,p) := |\{o|\exists(s,p,o) \in D\}|$$

□

To simplify the notation for cases where the triple $t = (s,p,o)$ and the reference element e are known we also use:

$$c(D,t,e) := \begin{cases} c_s(D,p) & \text{if } e = s \\ c_o(D,p) & \text{if } e = o \end{cases}$$

For the process of estimating pattern cardinalities we need these statistics available for fast lookup. We achieve this by computing and storing these values in advance in the database.

3.3.1 Estimating Cardinality of a Pattern

Query patterns are structures that can be represented using graphs. This feature allows us to analyze such structures by using basic graph theory. Basically, if a pattern consists of multiple graph components, its number of solutions is, at the end, the Cartesian product of the partial number of solutions for each individual graph component. Therefore, by estimating these components separately and multiplying their estimates, it is possible to achieve a reasonable estimation of the cardinality of the complete pattern. Note that this estimation is an upper and lower bound for the number of results. Therefore, in the rest

```
SELECT * WHERE {
    ?a p1 ?b .
    ?c p2 ?b .
    ?c p3 ?d
}
```

Listing 14: An example SPARQL query Q.

of this chapter, we analyze individual graph components to estimate the cardinality of a pattern.

The potential frequency of a pattern P is given by $O(|D|^{|P|})$, where $|D|$ is the size of the dataset (number of triples) and $|P|$ is the size of the pattern (number of triple patterns). This is a Cartesian product over the dataset ($D \times D \times \ldots \times D$), as every triple pattern from the query potentially can be bound to every triple in the database. This estimation results from the worst case where the query contains only variables instead of fixed values at subject, predicate and object positions and all variables are different. However, this case is rather unusual and can be disregarded as we restrict our estimations to patterns where predicates are bound.

According to this restriction, we can assure that the frequency of a pattern P is delimited as follows:

$$\#(P) \leq \prod_{\{p|\forall(s,p,o)\in P\}} f(D,p).$$

The right part of this formula represents a Cartesian product over the database, where the factors are estimations of single triples filtered by their predicates. Computing this estimation is straightforward if predicate frequencies are computed in advance. This function is an upper–bound for the real number of results of P.

However, patterns are usually given by a set of related single patterns. The set of solutions for a given pattern is obtained from the combination of results of each single triple pattern over the dataset. i.e., solutions that completely satisfy the pattern and not those that satisfy individual triple patterns. By applying the previous function we achieve general but not accurate cardinality information of the pattern P. Based on these arguments, we introduce now a novel idea to estimate the cardinality of a pattern over a dataset D.

Consider the query Q in Listing 14 and its graph representation in Figure 3.1.

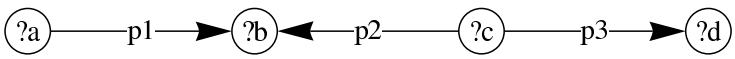


Figure 3.1: Pattern graph of the example query.

Every solution for this query contains one value for each variable. Every solution can be seen as a 4–tuple, where each value is assigned to a variable in the query. In general the complete set of results can be seen as an $(n \times m)$ matrix, where m is the number

of exported variables and n is the number of results. Each column contains all possible values for the corresponding exported variable of the query.

To continue our analysis, we choose one column of this matrix, e.g. the first, with values for the variable $?a$ and refer to it as the *root* element[3]. We observe that the total number of values assigned to the root equals the total number of results of Q. Additionally, in this column we observe the following facts:

- Distinct values

  There is a fixed number of distinct nodes in the data graph assigned to the *root* element. We refer to the number of distinct values for the root element as the *basis of the root*, written as $basis(root)$.

- Duplicate values

  There are results of the query that share the same value for the *root* element. It means that certain nodes in the data graph are used in the *root* element for a number of different results. We define the maximum number of duplicates over all different assignments for the root element as the *potential of the root*, written as $potential(root)$.

Using these two values, we can formulate an upper bound for the number of assignments for $?a$: $basis(?a) \times potential(?a)$. This upper bound also applies for the number of results for the query Q.

Notice that for the same set of results, selecting different root elements will produce different results for the basis and potential and, therefore, different estimations for the cardinality of the pattern.

Example 3.11 (Estimating Cardinality using Different Roots) *Consider the result set shown in Table 3.1 for the query in Listing 14. It is represented as a* (5×4) *matrix where each column represents the values assigned to one exported variable of the query, e.g. first column assigns to* $?a$*, second column to* $?b$*, etc. The number of solutions for the query equals the number of rows.*

Table 3.1: Result set of query Q. Each column represent assignments for one exported variable. Each row represents a solution for Q.

	$?a$	$?b$	$?c$	$?d$
1	a_1	b_1	c_1	d_3
2	a_1	b_1	c_2	d_4
3	a_1	b_1	c_3	d_4
4	a_2	b_1	c_4	d_5
5	a_3	b_1	c_4	d_5

[3] We assume, without loss of generality, that the variable $?a$ corresponds to the first column of the matrix. However, any other variable could be assigned.

Estimating the cardinality of the pattern regarding the root element ?a results in the following measures:

$basis(?a) = 3$

$potential(?a) = 3$

$cardinality(Q) = (3 \times 3) = 9$

As we want to show that different roots estimate different cardinalities for Q we compute this value for each possible root element. Table 3.2 shows the results for each root. The estimated cardinality changes when using different root elements.

Table 3.2: Estimating cardinality of Q by using different root elements.

	$basis(root)$	$potential(root)$	$cardinality(Q)$
$?a$	3	3	9
$?b$	1	5	5
$?c$	4	2	8
$?d$	3	2	6

In this model we estimate the frequency of the pattern regarding each possible root and finally selecting the minimum value. As all estimations are upper bounds for the real cardinality of the pattern, the minimum value is the closest estimate to the real number of results.

Up to now, we have shown an strategy to estimate the cardinality of a pattern by multiplying two measures obtained from the final set of results. Obviously, in a real setting this is not possible and we need heuristics to turn the problem around and estimate the basis and potential of the root without executing a query.

3.3.2 Estimating the Basis

We dedicate this section to present different strategies to estimate the basis of a root element.

We build our estimates upon statistics generated by using a particular RDF dataset and a given pattern. Notice that to achieve more accuracy in our estimations we require a higher level of granularity in our statistics. The following list exhibits estimations that can be used as upper bounds for the basis of the *root* element, where D is a dataset, P a query pattern and *root* the particular root element ($root \in \Xi(P)$):

- without any further information, $|D|$, i.e., the number of triples in the database
- if the root element is used as a subject in the query ($root \in \Sigma(P)$), the number of distinct subject values in the dataset, $|\Sigma(D)|$
- analogously, if the root element is used as an object ($root \in \Omega(P)$), the number of distinct object values in the dataset, $|\Omega(D)|$

- in case the root element is used in P both as a subject and as an object, $|\Sigma(D) \cap \Omega(D)|$
- for every predicate p that is used with the *root* ($(root, p, o) \in P \ \vee \ (s, p, root) \in P$), the frequency of the predicate p in the dataset D: $f(D, p)$
- $c(D, t, root)$, if the triple $t = (s, p, o)$ is an adjacent triple to the *root* element (see Section 3.3). It represents either the number of different subject nodes or the number of different object nodes used with the predicate p, depending on the position of *root* in t.

In the rest of the thesis, as our statistics are predicate-based we select the frequency of the predicate as the basis option to get an upper bound for the basis of the root (see Definition 3.10).

3.3.3 Estimating the Potential

In Section 3.3.1 we defined the *potential of a root* as the maximum number of results that share a common assignment for the root element. Consider the example query from Figure 3.1. Figure 3.2 shows subgraphs from an arbitrary fixed dataset D, emphasizing three assignments for the root element $?a$: a_1, a_2 and a_3.

Figure 3.2(a) shows a dataset subgraph around a_1 with one two–way branching, i.e., only two query results can be found where $a1$ is assigned to the root element $?a$. Figure 3.2(b) shows a dataset subgraph which contains a three–way branch for the triple $(?a, p1, ?b)$ around the node a_2 and a two–way branching for the triple $(?c, p3, ?d)$. Consequently, six results share the assignment a_2 for the root element $?a$. Finally, Figure 3.2(c) exhibits a more complex branching distribution around the node a_3 showing a two–way branch for the triple $(?c, p2, ?b)$ and a four–way branch for the triple $(?c, p3, ?d)$. According to this subgraph, the result set for the example query contains eight solutions where a_3 is assigned to element $?a$.

In basic graph theory this concept of branching is represented by two numbers that characterize every node of a directed graph: *indegree* and *outdegree*. *Indegree* and *outdegree* are defined in Definition 3.12.

Definition 3.12 *Let $G = (V, E)$ be a directed graph where V is a set of nodes and E is a subset of $(V \times V)$.*

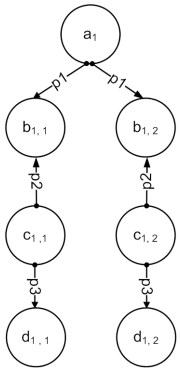
- *The* indegree *of a node $v \in V$ is defined as the number of edges that end at v:*

$$deg^-(v) := |\{(x, v) \in E\}|.$$

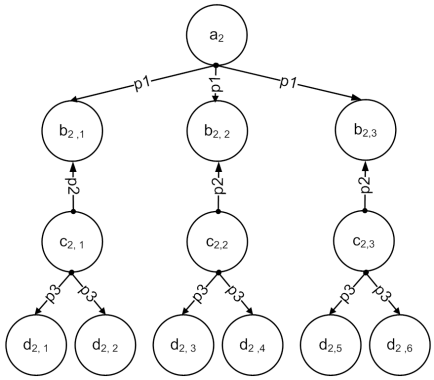
- *The* outdegree *of a node $v \in V$ is defined as the number of edges that start at v:*

$$deg^+(v) := |\{(v, x) \in E\}|.$$

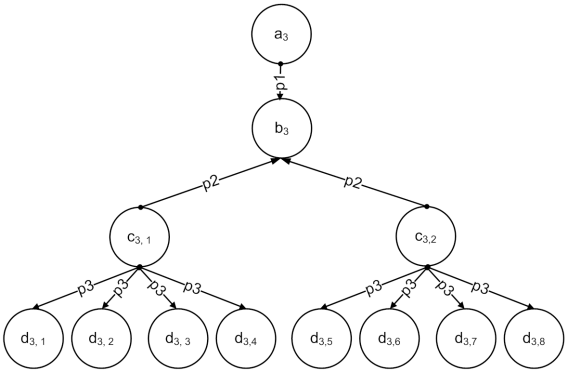
□



(a) Node a_1: One two–way branching, two results.



(b) Node a_2: One three–way branch and one two–way branch, six results.



(c) Node a_3: One two–way branch and one four–way branch, eight results.

Figure 3.2: Branching for specific assignments for the root element $?a$.

Applying these definitions to RDF, the indegree of a node n represents the total number of triples having n as an object. Similarly, the outdegree of a node n is the total number of triples that have n as their subject. For instance, in Figure 3.2(b), the node a_2 has an outdegree of 3 and the node $b_{2,3}$ has an indregree of 2.

Recall that we base our cost model on fixed predicates statistics. Therefore, in Definition 3.13, we qualify the indegree and outdegree according to predicate values. Intuitively, given a predicate p and a node n, the *fan of p at subject* corresponds to the predicate–specific outdegree of n, whereas the *fan of p at object* corresponds to the predicate–specific indegree of n. Table 3.3 shows a list of predicate–specific indegrees and outdegrees regarding Figure 3.2(c) for the nodes a_3, b_3, $c_{3,1}$ and $c_{3,2}$.

In Table 3.3, b_3 is an example of node with two indegrees (1 and 2) regarding p_1 and p_2 respectively. Note that b_3 does not represent a subject for any predicate. Thus, its

Table 3.3: List of predicate–specific values in & out degrees

	a_3		b_3		$c_{3,1}$		$c_{3,2}$	
	in	**out**	**in**	**out**	**in**	**out**	**in**	**out**
p_1	0	1	1	0	0	0	0	0
p_2	0	0	2	0	0	1	0	1
p_3	0	0	0	0	0	4	0	4

outdegree is zero regarding any possible predicate. Contrary, $c_{3,1}$ and $c_{3,2}$ are examples of nodes of which their indegrees are zero regarding any available predicate as they are never used as an object in the graph. In this case, both have outdegrees of 1 and 4 with respect to p_2 and p_3 respectively. Nodes $d_{3,1}$ to $d_{3,8}$ are not listed in Table 3.3 as they all share the same indegree value (1) regarding predicate p_3.

Definition 3.13 (Fan at subject, Fan at object) *Let D be an RDF database.*

- *For every combination of $s \in \Sigma(D)$ and $p \in \Pi(D)$, the* **fan at subject s of p** *is defined as:*
$$fan_s(D, s, p) := |\{(s, p, x) \in D\}|$$
- *For every combination of $o \in \Omega(D)$ and $p \in \Pi(D)$, the* **fan at object o of p** *is defined as:*
$$fan_o(D, o, p) := |\{(x, p, o) \in D\}|$$

□

Lemma 3.14 *For every predicate p:*

- *the sum over all fans at subject equals the frequency of p.*
- *the sum over all fans at object equals the frequency of p.*
- *the number of subjects of p that have a fan at subject higher than 0 equals the subject cardinality of p.*
- *the number of objects of p that have a fan at object higher than 0 equals the object cardinality of p.*

Definition 3.13 provides accurate statistical information about combinations between predicates and either subjects or objects. However, processing this information for each specific node may generate myriads of node–specific values, especially when computed over large datasets.

In general, the number of possible *fan of p at subjects* that can be generated from a dataset D can be defined as:

$$|\{fan_s(D, s, p)\}| \leq |\Sigma(D)| \cdot |\Pi(D)|$$

where $|\Sigma(D)|$ and and $|\Pi(D)|$ denote the number of subjects and predicates in D. This is an upper bound for the total number of subject statistics that need to be stored alongside the triple dataset. This is certainly a worst case where it is assumed that each subject is related with the complete set of predicates of D.

We can estimate the number of possible *fan of p at objects* in a similar way with the following upper bound:

$$|\{fan_o(D, o, p)\}| \leq |\Omega(D)| \cdot |\Pi(D)|$$

where, in this case, $|\Omega(D)|$ denotes the number of objects in D.

Putting all togheter, the total number of statistics (fan of p at subject and object) that could be generated from D can be denoted by:

$$|\{fan_s(D, s, p)\} \cup \{fan_o(D, o, p)\}| \leq |\Pi(D)|(|\Sigma(D)| + |\Omega(D)|)$$

To avoid such explosion of metadata we suggest the use of aggregate functions for the fan properties. In this way, we attain our goals of practicality and usefulness in our statistical measures.

We formalize this proposal in Definition 3.15 [65]. Both properties *fan at object* and *fan at subject* are estimated regarding their specific predicates by using the *max* aggregate function. Note that, as these two properties are predicate–specific, they can be straightforwardly preprocessed and stored alongside the predicate frequencies.

Definition 3.15 (Maximal fan at subject, Maximal fan at object) *Let D be an RDF database.*

- *For any predicate $p \in \Pi(D)$ the* **maximal fan at subject** *is defined as:*

$$fan_{max,s}(D, p) := max(\{fan_s(s, p) | \forall s \in \Sigma(D)\})$$

- *For any predicate $p \in \Pi(D)$ the* **maximal fan at object** *is defined as:*

$$fan_{max,o}(D, p) := max(\{fan_o(o, p) | \forall o \in \Omega(D)\})$$

□

Now that we have improved our predicate statistical measures, we turn to estimate the *potential of the root*. The idea is to traverse the query pattern starting at the root node and moving along the edges according to predicate counts. On each hop, we select one of the two maximal fans properties assigned to the corresponding edge. The selection of the fan is given by the direction we move over the edge. For instance, if we move from subject to object, we select the *maximal fan at subject*. Contrary, if we move from object to subject, we select the *maximal fan at object*.

After this process is accomplished, the *potential of the root* is obtained by multiplying all selected values for each predicate in the query pattern. The potential of the root is

defined as $potential_{max}(D, P, r)$, where D is the dataset, P is the pattern and $r \in \Xi(P)$ is the root element in P.

Example 3.16 (Estimating the maximal potential) *Let Q be a query as described in Listing 14 (Page 61). In this example we highlight how the potential estimation for the pattern $P(Q)$ is performed by computing the potential for every root element in the pattern. The estimation is computed using as dataset the union of the three graphs described in Figure 3.2.*

As shown before in Example 3.11, using different elements as roots may generate different estimates for the potential. Recall that the potential is defined in terms of the predicates and their relation with subjects and objects. The way they are selected is based on the position they have when traversing the pattern from an initial root element. Table 3.4 shows the fans of p at subject and fans of p at object for each predicate p in the pattern $P(Q)$.

Table 3.4: Maximal fans of p at subject and object over D.

predicate	$\mathbf{fan_{max,s}(D, p)}$	$\mathbf{fan_{max,o}(D, p)}$
$p1$	3	1
$p2$	1	2
$p3$	4	1

The potential of $P(Q)$ is computed while traversing the pattern. The graph representation of Q is shown in Figure 3.1 (Page 61). Table 3.5 shows the different potentials estimated by using each root element of the query pattern, i.e, ?a, ?b, ?c and ?d. During the traversal of the pattern each edge (each predicate denotes an edge) is covered and the maximal fan value is selected according to the direction of the edge. The maximal potential estimation for each root element is finally given by multiplying the selected values. As can be seen in Table 3.5, the potential can strongly vary according to root element that is chosen to estimate its value.

3.3.4 Dealing with Cycles

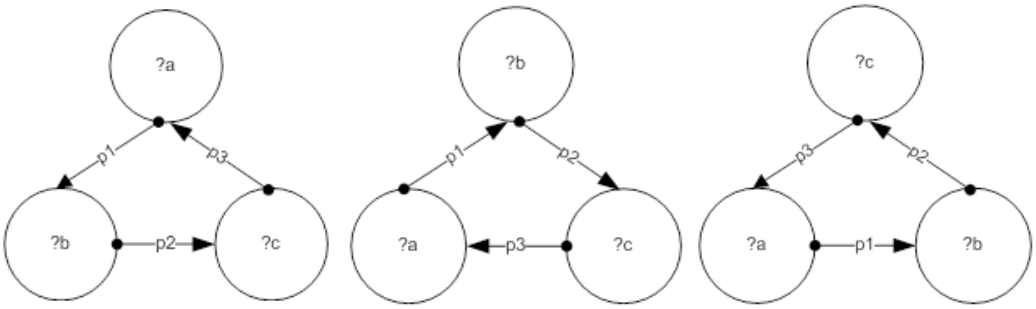
Cycles are an important factor to consider when estimating the potential of a pattern. Having a cycle in a pattern may generate infinite loops, in particular using a breadth–first algorithm, cycles prevent from specifying the order in which we move over the edges as the traversal of the pattern is done by level–order. We can get around this problem by considering cycle detection, i.e., a strategy that consider the structure of the pattern and the order in which it is processed.

To avoid infinite loops while traversing the graph, we hold a list of visited elements. Every time a graph component is traversed, a list of visited elements is created. This list is extended with every new visited element. Those elements in the list are afterwards excluded from any other traversal. When a new element is discovered, all its adjacent nodes are encountered, except those that have been already processed. For better understanding, we provide an analysis of a query pattern containing cycles in Examples 3.17 and 3.18.

Table 3.5: Estimating the potential of $P(Q)$ based on every root elements.

root	step	p1	p2	p3	potential
$?a$	1	**3**			
	2	3	**2**		
	3	3	2	**4**	
		3	2	4	**24**
$?b$	1	**1**			
	2	1	**2**		
	3	1	2	**4**	
		1	2	4	**8**
$?c$	1		**1**		
	2	**1**	1		
	3	1	1	**4**	
		1	1	4	**4**
$?d$	1			**1**	
	2		**1**	1	
	3	**1**	1	1	
		1	1	1	**1**

Example 3.17 (Cycles in query patterns) *Let P be a query pattern containing a cycle where $P = (?a, p_1, ?b), (?b, p_2, ?c), (?c, p_3, ?a)$. Figure 3.3 shows different representations for this pattern regarding each possible root element.*



(a) root element $?a$ (b) root element $?b$ (c) root element $?c$

Figure 3.3: Graphical representation of pattern P using three different elements as root.

Processing the pattern without cycle detection would yield that the algorithm iteratively traverses the edges incurring in infinite loops. Table 3.6 shows an excerpt of the traversal tree using the root element $?a$. On each level, all edges adjacent to the element $?a$ are discovered and all adjacent nodes are listed as possible next elements to visit.

Table 3.6: Infinite loops in $P(Q)$ based on root element $?a$.

element	adjacent elements	edges to visit
$?a$	b, c	$p1, p3$
$?b$	a, c	$p1, p2$
$?c$	a, b	$p3, p2$
$?a$	b, c	$p1, p3$
...	...	...

Applying cycle detection we avoid this problem and allows us to estimate the potential of the roots. As described in Example 3.16, our approach traverses the patterns and multiplies the fans for each predicate for all root elements. Using the fans described in Table 3.4, this would yield that $potential(?a) = 3$, $potential(?b) = 1$, *and* $potential(?c) = 24$, *where the selected potential would be 24.*

However, we can also notice that the potential for this pattern regarding the root element $?a$ *is upper bounded by multiplying the fans between* $(?a, p_1, ?b)$ *and* $(?b, p_2, ?c)$ *i.e.,* $3 \times 1 = 3$. *Evidently, the only possible value that can be reached by traversing the predicate* p_3 *of the triple pattern* $(?c, p_3, ?a)$ *is the initial value assigned to the node* $?a$ *of the triple* $(?a, p_1, ?b)$[4]. *Similarly, using the root element* $?b$, *the potential would be upper bounded by multiplying the fans of* $(?b, p_1, ?a)$ *and* $(?b, p_2, ?c)$, *i.e.,* $1 \times 1 = 1$ *since* $(?c, p_3, ?a)$ *can only take those values assigned to the node* $?a$.

□

The previous example provides in a simple yet illustrative way how cycles are treated using this approach. In the following example we analyze a query pattern containing multiple cycles.

Example 3.18 (Cycles in query patterns: A more complex example) *Assume a pattern* P_2 *containing mulitple cycles as described in Figure 3.4.*

$$P_2 = (?a, p_1, ?b), (?a, p_2, ?c), (?b, p_3, ?c), (?c, p_4, ?d), (?c, p_5, ?b), (?d, p_6, ?a)$$

Table 3.7 describes a traversal of the graph. Starting from the node $?a$, *we look at all its adjacent edges, e.g.* p_1, p_2, p_6 *and its potential next elements i.e.,* $?b, ?c, ?d$. *We visit all adjacent edges and store them in a list of visited edges. From this list, they are excluded from further traversal in the subsequent evaluation. This exclusion can be seen at the second row with the element* $?b$. *The adjacent edges are* p_1, p_3, *and* p_5, *however, our strategy only visits* p_3, p_5 *as* $p1$ *has been visited already. The process continues in the same way until the traversal of the graph is completed.*

□

[4]We assume that the dataset contains no duplicates.

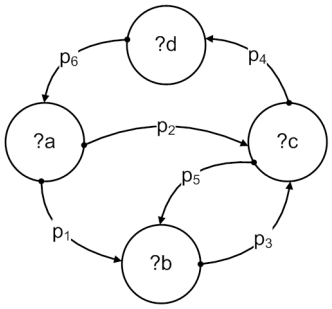


Figure 3.4: Query pattern containing multiple cycles.

Table 3.7: Dealing with cycles by traversing the graph.

element	next elements	adjacent edges	visited edges
$?a$	$?b, ?c, ?d$	p_1, p_2, p_6	p_1, p_2, p_6
$?b$	$?a, ?c$	p_1, p_3, p_5	p_3, p_5
$?c$	$?a, ?b, ?d$	p_2, p_3, p_4, p_5	p_4
$?d$	$?a, ?c$	p_4, p_6	-

Examples 3.17 and 3.18 showed that our strategy is a suitable strategy to avoid infinity loops and also highlighted that breaking cycles does not always guarantee a more accurate cardinality estimation since it also depends on the selected root element and its the adjacent nodes. Thus, in this approach cycle detection is applied and, similarly to cycle–free patterns, all edges are traversed to estimate the potential of a given pattern.

From these analysis and based on Definition 3.15 we define the maximal potential of a pattern as follows.

Definition 3.19 (Maximal potential) *Let D be a dataset and P a pattern. Let $r \in \Xi(P)$ be the root. The* **maximal potential** *for r is defined as:*

$$potential_{max}(D, P, r) := \prod_{t=(s,p,o)\in P} \begin{cases} fan_{max}(D, t, s) & , \text{ if } d(s, r) < d(o, r) \\ fan_{max}(D, t, o) & , \text{ if } d(o, r) < d(s, r) \end{cases},$$

where $d(u, v)$ is the distance between the elements u and v of $\Xi(P)$, i.e. the length of the shortest path from u to v in P.

□

3.3.5 Constants in Patterns

There is another fact to consider when estimating the cardinality of a pattern: constant elements in triple patterns. During the estimation of the basis and potential of a root,

we traverse the pattern and gather statistics regardless element values. i.e., we do make distinction between constants and variables.

Generally, we may think that a constant in the pattern could be the best root candidate as the constant remarkably restricts the set of results to those solutions containing that value. However, it is not guaranteed to be the best choice as we explain in Example 3.20.

Example 3.20 (Constant elements) *Let Q be a query with a constant element a where $Q = (a, p_1, ?b), (?b, p_2, ?c)$ and D a dataset as described in Figure 3.5.*

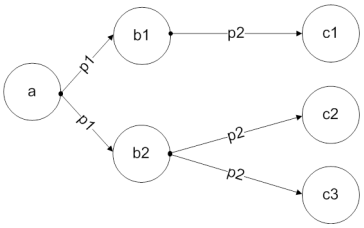


Figure 3.5: Example dataset D.

Executing query Q over D would returns a set of results with values assigned to the variables contained in the query. Assume that we get not only those values assigned to variables but the subgraphs from the dataset that match with the query pattern. We show in Table 3.8 a tabular representation of these results.

Table 3.8: Assignments for all variables of Q. Constant are included to represent the complete set of solution for the pattern.

a	$p1$	$?b$	$p2$	$?c$
a	$p1$	$b1$	$p2$	$c1$
a	$p1$	$b2$	$p2$	$c2$
a	$p1$	$b2$	$p2$	$c3$

To shown that constant values are not always the best choice to estimate cardinality of a pattern, we evaluate the basis and potential for the constant a and variables b and c as roots. Evidently, the basis for the constant root a is 1, because a is the only distinct value available in the results set and the potential regarding the same root element is 4 as can be seen in Equation 3.20. Thus, the cardinality estimation for the pattern of Q regarding the root element a equals 4.

$$potential_{max}(D, P, a) = fan_{max,s}(D, p1) \times fan_{max,s}(D, p2) = 2 \times 2 = 4$$

Now, we turn to estimate the potential for ?b:

$$potential_{max}(D, P, ?b) = fan_{max,o}(D, p1) \times fan_{max,s}(D, p2) = 2 \times 1 = 2.$$

The basis for ?b according to the predicate cardinalities equals the subject cardinality of $p2 = 2$*. Therefore, the cardinality estimation for Q regarding the root element b is also* 4*.*

Finally, the basis for ?c equals the object cardinality of $p2 = 3$ *and the potential equals 1 (see Equation 3.20). In this case, the cardinality estimation for Q is:* $3 \times 1 = 3$*.*

$$potential_{max}(D, P, ?c) = fan_{max,o}(D, p2) \times fan_{max,o}(D, p1) = 1 \times 1 = 1.$$

Table 3.9: Cardinality estimation regarding every root elements and the constant a.

root	*basis*	*potential*	*cardinality*
a	1	4	4
$?b$	2	2	4
$?c$	3	1	3

All estimated cardinalities in Table 3.9 are upper bounds for Q. However, the estimated cardinality is bigger if we use the constant element instead of the variable ?c as root, and therefore, a is not the best root.

A constant value represents a restriction in the pattern that distinguishes the set of solutions for the given query. Estimating the fraction of results constrained by the constant value is possible applying Definition 3.10. We motivate this argument in Example 3.21.

Example 3.21 (Constant values in query patterns) *Let* q_1 *and* q_2 *be two query patterns where* $q_1 = (?x, p_1, ?y)$ *and* $q_2 = (?x, p_1, b1)$ *and* $b1$ *is a constant.*

Matching both queries against the dataset in Figure 3.5 returns the set of results shown in Tables 3.10 and 3.11.

Table 3.10: Results of q_1

$?x$	$p1$	$?y$
a	$p1$	$b1$
a	$p1$	$b2$

Table 3.11: Result of q_2

$?x$	$p1$	$b1$
a	$p1$	$b1$

Obviously, the result in Table 3.11 is a subset of the results contained in Table 3.10. Table 3.11 contains one result from Table 3.10 where the variable ?y is assigned to the constant value $b1$*. We can see that the constant value in* q_2 *selects a fraction of the results of* q_1*. This fraction is given by* $\frac{1}{variety}$*, where variety is the number of different values of ?y.*

In our approach we can estimate the *variety* by looking at predicate properties. Specifically, we look at the subject or object cardinalities of the adjacent triples as the most suitable measures to estimate these cases (see Definition 3.10).

- If the element with the constant is used as the subject of an adjacent triple pattern with the predicate p, $c_s(D, p)$ is an estimation for the variety of the element.
- If the element with the constant is used as the object of an adjacent triple pattern with the predicate p, $c_o(D, p)$ is an estimation for the variety of the element.

The variety represents the number of different values assigned to a given element. Using subject and object cardinalities to estimate the variety of an element provides an upper bound for the real variety of that element. For instance, having a constant element adjacent to three different triples, assuming statistical independence, it is possible to estimate three different varieties. Each of these estimations is a upper bound for the constant element and we could also assure that the real variety of this element cannot be larger than the smallest estimated value. The smallest variety estimation represents the cardinality of a subject or object of one adjacent triple to the constant element, i.e., represents a fixed limit for the number of different subjects or objects that can take place in that triple. The statistical mean of all possible varieties may be also used as an estimation of the real variety, however this aggregation function is guaranteed to be larger than the smallest estimated variety. Under the assumption of statistical independence, every constant in the query pattern can be handled in the same way, because every constant selects a fraction of the results whereby a combination of constants selects the multiplied fraction.

Example 3.22 (Constant elements: Refining the estimation) *Let p be a query pattern where $p = (a_1, p_1, ?b), (a_1, p_2, ?c), (a_1, p_3, ?d)$ and a_1 is a constant. Consider a datset D dataset as described in Table 3.12. Computing the real cardinalities of all different constants would generate a blow of statistical data as explained in Section 3.3.3. However, using* subject cardinality of *p (see Definition 3.10) could be used to estimate the variety of an specific constant element for each adjacent triple. The estimated varieties regarding the constant element a_1 are shown in Table 3.13. According to our assumptions, all values represent upper bounds for the real variety of the element a_1 over each adjacent triple whereby the complete set of results for the given pattern is restricted. Therefore, the minimal estimated variety of a_1 is the most accurate estimation, i.e., $c(D, t, a_1) = 2$, where $t = (a_1, p_1, ?b)$.*

Finally, Definition 3.23 describes a method to estimate the fraction of results generated if the pattern contains constant elements.

Definition 3.23 (Constant elements fraction) *Let D be a dataset and P a pattern. The* **constant elements fraction** *can be estimated with:*

$$fraction(D, P) := \prod_{constant\ elements\ e \in P} \frac{1}{min(\{c(D, t, e) | t \in \Theta(P, e)\})}.$$

□

Table 3.12: Dataset D.

s	p	o
a_1	p_1	b_1
a_2	p_1	b_2
a_1	p_2	c_1
a_2	p_2	c_2
a_3	p_2	c_3
a_1	p_1	d_1
a_2	p_1	d_2
a_3	p_1	d_3
a_4	p_1	d_4

Table 3.13: Varieties regarding p.

predicate	$c(D, t, a_1)$
p_1	2
p_2	3
p_3	3

3.3.6 Cardinality Estimation and Cost Model

In Section 3.3.2 and Section 3.3.3 we provided estimations for the basis and potential, measures that can be used to estimate the cardinality of a pattern.

Up to now, we have defined our measures as upper bounds for the cardinality estimation. However, we extend our statistical measures to other aggregate functions namely, *min, mean and median.* We assume that the *mean* and *median* functions represent the average cases and we base our evaluation on these measures.

We provide general functions to identify our measures. For instance, $fan_M(D, t, e)$ is used to denote the aggregated predicate fans, where M can be one of *min*, *mean*, *median* or *max* and t is an adjacent triple to the element e. Similarly, $potential_M(D, P, r)$ represents the estimation of the potential for the root r in the pattern P, using the results estimated by using $fan_M(D, t, e)$. In Definition 3.24, we summarize our proposal based on the aggregate functions min, mean, median and max.

Definition 3.24 (Estimated cardinality) *Let D be a dataset and P a pattern. For an aggregation method M (one of min, mean, median and max), the* **estimated cardinality** *of P in D can be calculated using:*

$$estimate_M(D, P) := fraction(D, P) \times M(\{f(D, p) \times potential_M(D, P, r) | r \in \Xi(P)\}).$$

□

From Definition 3.24, we can now derive a cost function. Its main goal is to indicate the cover that achieves the fastest execution of a query. We strive for a cost model that has a low error in the average case. To that end, the cost model evaluates the statistical properties for the predicates in the dataset. The model takes into account the query pattern and the patterns of the materialized views defined in the system.

There are mainly three tasks that influence the query execution time in RDFMatView.

- retrieve the cover
- compute the residual
- join both result sets

We propose to model these tasks by finding the cardinality of the cover and residual pattern of the query. Additionally, we estimate the number of comparisons required between both results during the join phase.

Retrieving one result of the covered pattern is faster than computing one result of the residual pattern. Recall that the covered pattern retrieves its solutions by implementing joins on materialized queries stored previously in the database system. On the other hand, the residual pattern is executed by a SPARQL query processor. Thus, processing the residual usually requires a number of self joins on a large triple table.

We model these differences by assigning weights to the estimated cardinalities. The idea is to punish the residual processing. Definition 3.25 shows the CardiOS cost model proposed for RDFMatView:

Definition 3.25 (CardiOS:Cost Model) *Let D be a database, Q a query, C a cover for the query Q and I a set of participating indexes in C. Let w_c and w_r be two real values for weighting the covered and the residual estimate respectively with $w_c \geq 0$ and $w_r \geq 0$ and $w_c + w_r = 1$.*

For a method M (one of min, mean, median, max), the estimated weighted cost *of executing Q using the cover C is defined as:*

$$wCost_M(Q, C) := e_c \times e_r \times \frac{w_c \times e_c + w_r \times e_r}{e_c + e_r},$$

where $e_c = estimate_M(D, C)$ and $e_r = estimate_M(D, R)$.

□

3.4 SPOracle:A Join–Statistical Cost Model for SPARQL

In Section 3.2 we described SyCoM (see Definition 3.6), a cost model based on a special form of selectivity. Making strong assumptions on query processing this simple model estimates the selectivity of a cover. In Section 3.3 we introduced CardiOS, a model based on predicate–statistics that estimates the cardinality of a pattern. The former model bases its estimations on covered patterns and disregards those patterns not covered by indexes. The second model takes into account the complete set of patterns but disregards statistical metadata generated from the set of indexes. In this section, we propose a third cost model that combines both sources of statistics, i.e., indexes and datasets. Specifically, we apply index statistics to estimate the cardinality of covered patterns and predicate statistics to estimate the cardinality of patterns that are not covered by indexes.

Definition 3.24 showed how to estimate the cardinality of a given pattern by looking only at predicate statistics generated from the original RDF dataset. Note that these statistics

disregard all information stemming from the preprocessed indexes. Now, we introduce a method to estimate the cardinality specifically of the covered part of a query by using metadata gathered from this set.

As stated in Chapter 2, to process a query using RDFMatView, we analyze which patterns of the query can be covered by a set of indexes. Our indexes are basically SPARQL queries materialized as traditional relational tables, which are joined to compute the entire cover. Therefore, it makes sense to use estimates for join sizes proposed in relational database theory. Specifically, we need to estimate the size of a join among multiple tables with multiple join attributes.

In Section 3.1 we described how join sizes can be estimated. Basically, this process requires three steps: First, to multiply the sizes of each participating relation. Second, for each attribute that appears at least twice in the join, we take its number of different values in the relations, exclude the lowest value, and multiply the rest of them. Finally, we compute the ratio between the two factors.

To compute the size of a join we need the frequency of each index ($\#(i)$) in the cover and the variety of its contained attributes, i.e., $variety(i, a)$ where a is an attribute of i.

Both parameters can be obtained from the set of indexes by scanning each index and counting the number of tuples and the number of different values assigned to each attribute a as well. These values are not estimates but real values.

Example 3.26 (Estimating cover cardinality by estimating join sizes) *Let Q be a query as described in Listing 15 and $index_1$, $index_2$, $index_3$ indexes for Q as described in Listings 16 to 18.*

```
SELECT * WHERE {
    ?a p1 ?b .
    ?a p2 ?c .
    ?a p3 ?d .
    ?d p4 ?e .
    ?d p5 ?f .
    ?f p6 ?g .
    ?g p7 ?h .
    ?g p8 ?i .
}
```

Listing 15: SPARQL query.

```
SELECT * WHERE {
    ?a p1 ?b .
    ?a p2 ?c .
    ?a p3 ?d .
}
```

Listing 16: RDFMatView $index_1$.

```
SELECT * WHERE {
    ?a p3 ?d .
    ?d p4 ?e .
    ?d p5 ?f .
}
```

Listing 17: RDFMatView $index_2$.

```
SELECT * WHERE {
    ?d p5 ?f .
    ?f p6 ?g .
}
```

Listing 18: RDFMatView $index_3$.

At index processing time, indexes are materialized and statistics are stored alongside in the database system. Examples of these statistics are described in Table 3.14.

Table 3.14: Index statistics.

index	$frequency$	$variety_a$	$variety_d$	$variety_f$
$index_1$	15	6	8	-
$index_2$	25	5	9	10
$index_3$	40	-	5	8

Analyzing one cover that contains all three indexes, the join attributes among all materialized indexes are denoted by the variables ?a, ?d, and ?f.

We propose to estimate the cardinality of the covered part of the query based on estimations of join size. Since our indexes are given in the form of relational tables,this strategy can be applied by looking at the index properties.

The idea is to estimate the size of the join by computing the ratio between the frequency of the participating indexes and the variety of the join attributes as described in [33]. In this example the estimation of the size of the join would be given by:

$$size(index_1 \bowtie index_2 \bowtie index_3) := \frac{15 \times 25 \times 40}{6 \times 8 \times 9 \times 10} \approx 3$$

The final number of estimated results is given by joining the estimated results of the covered and residual parts of the query. In this example, the cardinality of the residual pattern $(?g, p7, ?h)(?g, p8, ?i)$ needs to be estimated and the results joined with those obtained from the indexes.

□

In the CardiOS model introduced in Section 3.3 the pattern cardinality is achieved by computing the basis and potential. These two measures are based solely on predicate statistics gathered from the dataset. In Example 3.27 we provide a a comparison of CardiOS and SPOracle and highlight the advantages of using index statistics.

Example 3.27 (SPOracle vs. CardiOS: Using exact vs. approximated statistics) *Let Q be a query as described in Listing 14 and $index_1$, $index_2$ indexes for Q as described in Listings 19 and Listing 20. The estimation is computed using as dataset the union of the three graphs described in Figure 3.2. Statistics for the indexes are described in Table 3.15.*

```
SELECT * WHERE {
    ?a p1 ?b .
    ?c p2 ?b .
}
```

Listing 19: RDFMatView $index_1$.

```
SELECT * WHERE {
    ?c p2 ?b .
    ?c p3 ?d .
}
```

Listing 20: RDFMatView $index_2$.

Estimating the size of cover by using the join size estimation would return the following result:

$$size(index_1 \bowtie index_2) := \frac{7 \times 16}{7 \times 6} \approx 3$$

Table 3.15: Index statistics.

index	$frequency$	$variety_c$	$variety_b$
$index_1$	7	7	6
$index_2$	16	7	6

On the other hand, estimating the basis and potential regarding the element ?a would return the following estimation:

$$basis(?a) \times potential(?a) = 6 \times 24 = 144$$

To verify which of this strategies is more accurate to estimate the cardinality of the pattern, we compare the errors of these two estimations regarding the real number of results of the query according to the error formula $error_x = \frac{|estimate_x - real|}{real}$.

Table 3.16: Error comparison.

strategy	*error*
join size	0.81
cardinality	8.0

From these results we can assert that join size estimation offers a higher degree of accuracy than the basis and potential to estimate the cardinality of a pattern.

Definition 3.28 shows the final estimate for join size of the covered part of a query.

Definition 3.28 (Estimated cardinality of a cover) *Let* C *be a cover and* I *be a set of indexes such that* $\forall i \in I,\ i \in C$ *. Let* A *be the set of join attributes among indexes* $i \in I$*. The* **size of the join** *among all* $i \in I$ *can be estimated with:*

$$spo\_card(C, I) := \frac{\prod_{\forall i \in I} \#(i)}{\prod_{\forall i \in I} variety(i, x_j)}$$

where $x_j \in A$ *and* $v(i, x_j) \geq v(k, x_j)\ \forall k \in I$ *.*

□

Definition 3.28 describes a strategy to estimate the cardinality of a cover, i.e., those patterns of a query that can be processed by using materialized indexes. However, we strive for a cost model that provides estimations for the complete query pattern, including those patterns that cannot be covered by indexes. Definition 3.24 offers a suitable solution for this goal. Therefore, in Definition 3.29 we define SPOracle, as a cost model that combines the cardinality estimation of a cover and the pattern cardinality estimation.

As in CardiOS, our current model considers the retrieval of the cover, execution of the residual patterns, and the join operation of both results. We favored the processing time of the covered patterns by applying weights to the estimated cardinalities. As stated before, retrieving results of the covered pattern is faster than computing the residual pattern. Results of covered patterns are obtained by joining materialized queries stored in the underlying database system and residual patterns are executed on demand using a SPARQL query processor.

Definition 3.29 (SPOracle:Cost Model) *Let D be a database, Q a query, C a cover for the query Q and I a set of participating indexes in C. Let w_c and w_r be two real values for weighting the covered and the residual estimate respectively with $w_c \geq 0$ and $w_r \geq 0$ and $w_c + w_r = 1$.*

The estimated weighted cost *of executing Q using the cover C is defined as:*

$$SPOracle(Q, C) := e_c \times e_r \times \frac{w_c \times e_c + w_r \times e_r}{e_c + e_r},$$

where $e_c = spo\_card(C, I)$ and $e_r = estimate_{mean}(D, R)$. □

Notice that weights in our model are used only to distinguish covered from residual results by the time it takes to retrieve one of them. In Section 3.5 we provide an evaluation of this model using different weights. Further investigation concerning to select an optimal pair of weights for the covered and residual patterns regarding specific characteristics of the workload is an important topic for future research.

3.5 Evaluation

3.5.1 Dataset and Queries

Our tests are performed in the context of the same benchmarks as described in Chapter 2: Berlin SPARQL Benchmark [10] and SPARQL Performance Benchmark (SP^2B) [80]. Using the general setup of queries and indexes defined in Chapter 2 we first validate our simple model in Section 3.5.2 by showing the correlation of the estimated selectivity with the real selectivity for each cover. Afterwards, in Section 3.5.3 we evaluate the accuracy of the second and third model by comparing the estimated values to the real number of results for a specific pattern and by analyzing errors of the resulting estimations. Section 3.5.4 shows the evaluation of CardiOS and SPOracle models using different pairs of weights. In Section 3.5.5 we show how accurate our models are by evaluating the real runtimes of optimized queries. Finally, in Section 3.5.6 we compare the accuracy of all models by regression analysis.

3.5.2 Accuracy of Selectivity Estimation: Evaluating Covers

In Chapter 2 we have shown that using RDFMatView achieves savings in processing time when using materialized views as indexes. Our results showed that, according to

the selected rewriting method, the performance of query processing remarkably varies. However, they also evidenced that improvements in query processing strongly depend on a proper selection of the cover used to execute the query. Here, we show how the use of the SyCoM model (defined in Section 3.2) to evaluate candidate indexes provides a reasonable way to know how the covers perform at processing time.

Figure 3.6 illustrates the selectivity (estimated and real) for each cover over all test queries[5]. Clearly, predicted values for selectivity are, on average, far from being accurate regarding real values. However, this model was not designed to provide accurate values. Our estimations are purely abstract and we only strive for information to predict an optimal cover at processing time. In this sense, the model satisfies this expectation to some degree. We can see that there exists a rough correlation among the values over the covers. For instance, Figures 3.6(a), 3.6(b) and, 3.6(d) show a strong correlation between real and estimated selectivity whereas in Figure 3.6(c) (covers 1, 4, and 7), Figure 3.6(e) (covers 1 and 2) and, Figure 3.6(f) (covers 1, 4, and 5), the values differ from each other significantly.

Covers are only a part of the query pattern that can be processed using indexes. At this point we found the most important drawback of the model. It only considers the covered part of the query. Residual patterns are completely disregarded and may lead to significantly inaccurate estimations of processing time. This especially happens if those residual patterns have a large number of solutions in the data graph or the number of residual patterns requires a large number of joins in the triple table. We provide a runtime evaluation of this model further in Section 3.5.5.

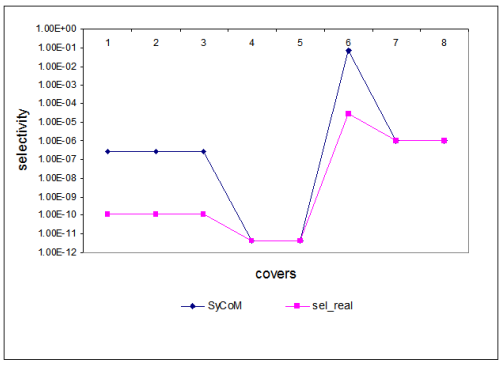
3.5.3 Estimating Cardinality with CardiOS and SPOracle

We evaluate the estimation of pattern cardinality regarding *predicate–statistics* and *join size*, i.e., cardinalities are estimated using the CardiOS and SPOracle models. In this context, we compare estimated to real cardinality of the same pattern. Our measurements are done over our test queries, gathering the cardinality values for their resulting sets of covered and residual patterns. For CardiOS we evaluate the mean and median function introduced in Section 3.3 and described in Definition 3.24. SPOracle is evaluated using the cover join size estimation introduced in Section 3.4, Definition 3.28.

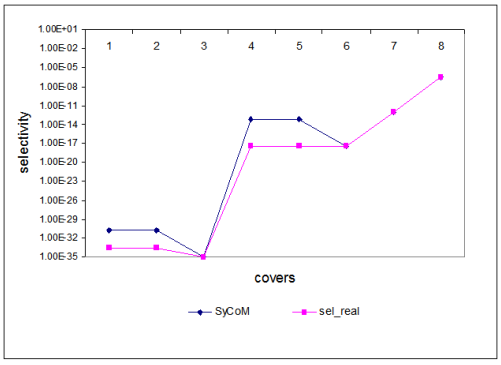
Figure 3.7 shows estimated and real cardinality values for both covers and residuals of $query_1$. The x-axis ranges over all covers computed for query1 and the y–axis represents the cardinality for each cover.

Our results in Figure 3.7 and Table 3.17 show that for the covered and residual patterns from the covers of query1 our estimations correlate with the real number of results. Estimations of the residual parts evidence a high estimation accuracy. Figures 9 to 12 in Appendix C Section 6 show the charts for the remaining queries 2, 3, 5 and 6 over both benchmarks. All these charts display estimates in different degrees of accuracy as well as a clear correlation between estimated and real values.

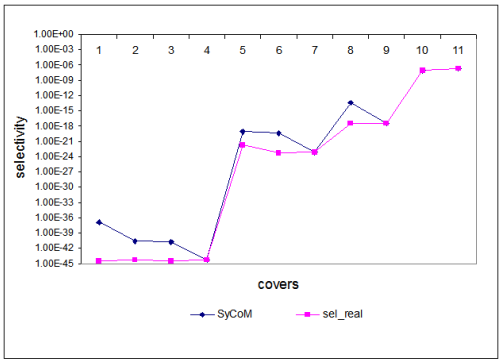
[5]In the graphics, the use of solid lines does not indicate that the plotted data is continuous but visually simplifies to notice the correlation between estimated and real values.



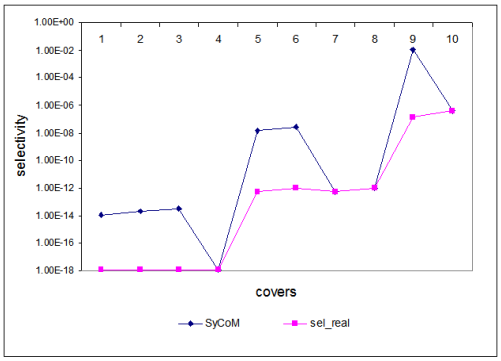
(a) Selectivity covers query 1.



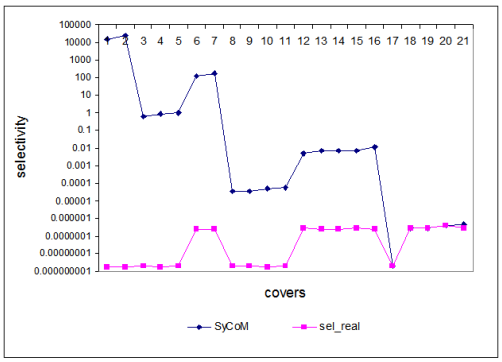
(b) Selectivity covers query 2.



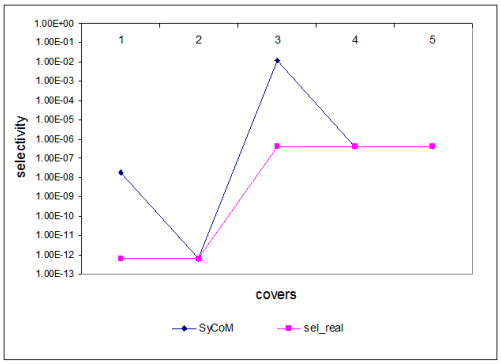
(c) Selectivity covers query 3.



(d) Selectivity covers query 4.

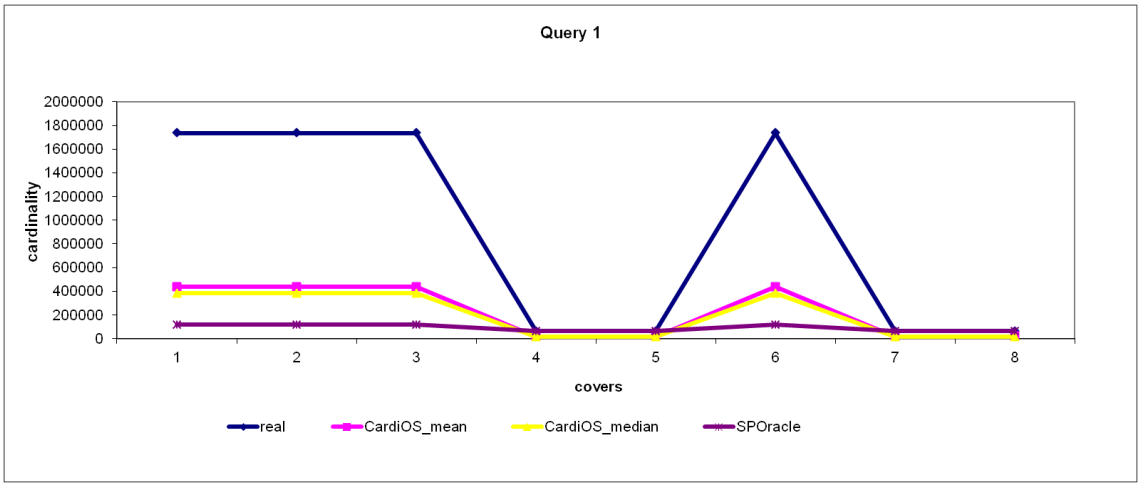


(e) Selectivity covers query 5.

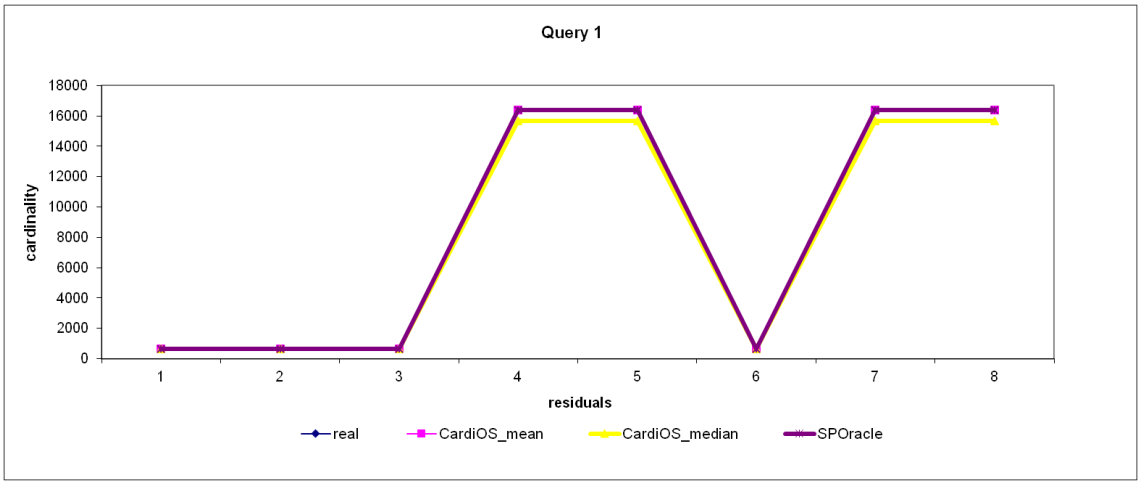


(f) Selectivity covers query 6.

Figure 3.6: Evaluation of selectivity estimation.



(a) Estimation on the covered parts.

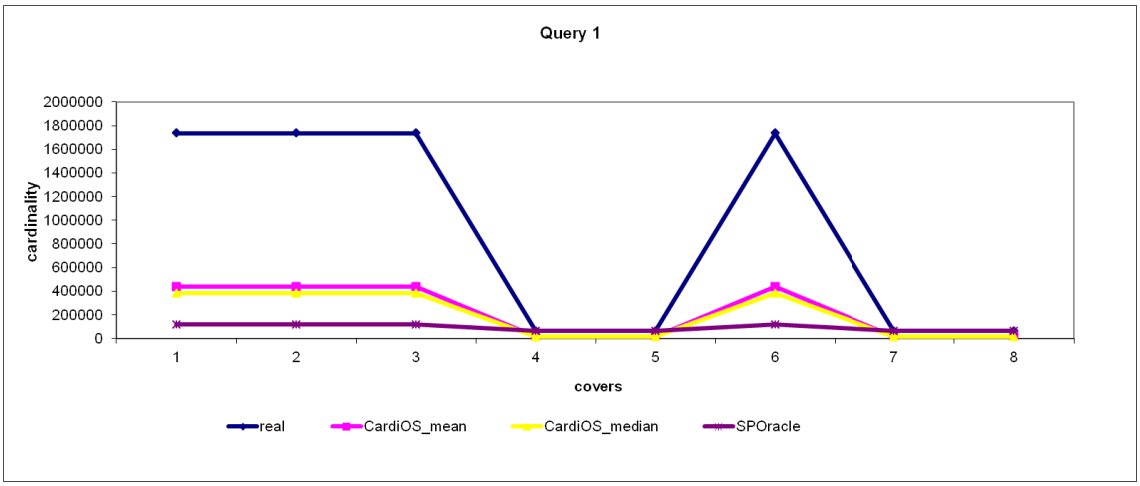


(b) Estimation on the residual parts.

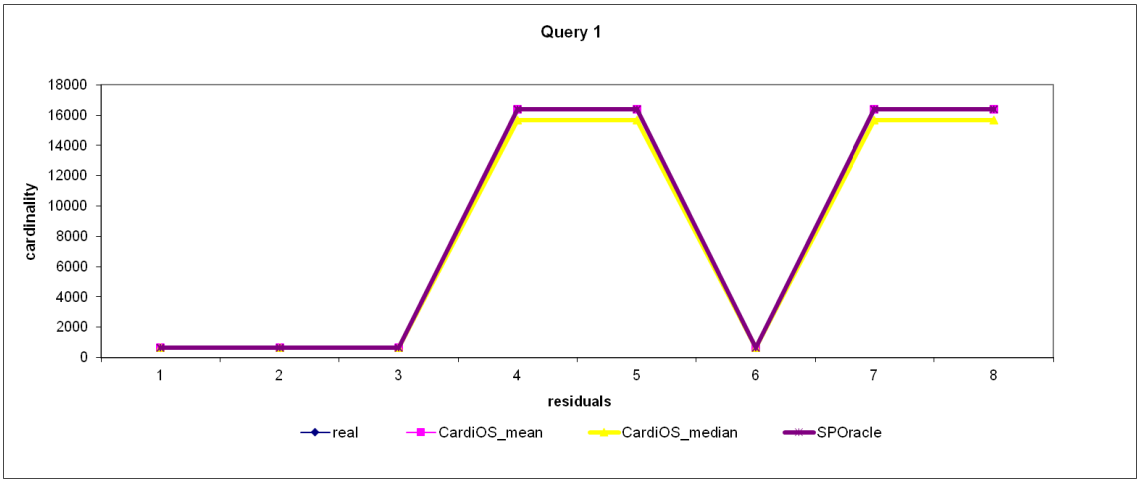
Figure 3.7: Evaluation of estimation on covers in query 1 (BSBM).

Results of query4 over the SP^{2}B benchmark, shown in Figure 3.8 differ from those over the Berlin Benchmark. This fact can be attributed to the inclusion of constant elements in the queries. For instance, query4 contains a constant element bench:Inproceedings with the rdf:type predicate, query5 the element bench:Article with the same predicate rdf:type and query6 the element foaf:Document with the rdfs:subClassOf predicate. The influence of these elements is estimated by our estimation strategy inaccurately.

For our test queries, the models CardiOS (using the mean function) and SPOracle provide the most accurate estimates of the pattern cardinality. In general, the mean function definitively achieves a higher degree of accuracy than the median function when



(a) Estimation on the covered parts.



(b) Estimation on the residual parts.

Figure 3.7: Evaluation of estimation on covers in query 1 (BSBM).

Results of query4 over the SP^{2}B benchmark, shown in Figure 3.8 differ from those over the Berlin Benchmark. This fact can be attributed to the inclusion of constant elements in the queries. For instance, query4 contains a constant element bench:Inproceedings with the rdf:type predicate, query5 the element bench:Article with the same predicate rdf:type and query6 the element foaf:Document with the rdfs:subClassOf predicate. The influence of these elements is estimated by our estimation strategy inaccurately.

For our test queries, the models CardiOS (using the mean function) and SPOracle provide the most accurate estimates of the pattern cardinality. In general, the mean function definitively achieves a higher degree of accuracy than the median function when

Table 3.17: Cardinality estimation of query1 using CardiOS and SPOracle.

	$\mathbf{CardiOS_{mean}}$	$\mathbf{CardiOS_{median}}$	$\mathbf{SPOracle_{join}}$	**real**
$cover_1$	436,613	384,977	117,224	1,735,232
$residual_1$	666	666	666	666
$cover_2$	436,613	384,977	117,224	1735232
$residual_2$	666	666	666	666
$cover_3$	436,613	384,977	117,224	1,735,232
$residual_3$	666	666	666	666
$cover_4$	17,750	16,382	65,528	65,528
$residual_4$	16,381	15,651	16,381	16,382
$cover_5$	17,750	16,382	65,528	65,528
$residual_5$	16,381	15,651	16,381	16,382
$cover_6$	436,613	384,977	117,224	1,735,232
$residual_6$	666	666	666	666
$cover_7$	17,750	16,382	65,528	65,528
$residual_7$	16,381	15,651	16,381	16,382
$cover_8$	17,750	16,382	65,528	65,528
$residual_8$	16,381	15,651	16,381	16,382

used in the CardiOS model. For covered patterns, SPOracle usually outperforms CardiOS independently from its aggregation functions. However, to determine which model offers an overall more accurate estimation of the query pattern we require further evidence. Therefore, we perform analysis of errors in the following section.

Error Analysis

We estimate error values according to estimations of the cardinality for all covers and residual patterns of each test query. Values are expressed in percent using the following formula:

$$error_x = \frac{|estimate_x - real|}{real},$$

where x is either $CardiOS_{mean}$, $CardiOS_{median}$ or $SPOracle$.

Table 3.18 suggests that the mean cost estimation is more accurate than the median cost estimation. However, to deduce which between mean and join estimates is more accurate, standard deviation of the errors are given in Table 3.19. Mean estimates are slightly more robust for each query.

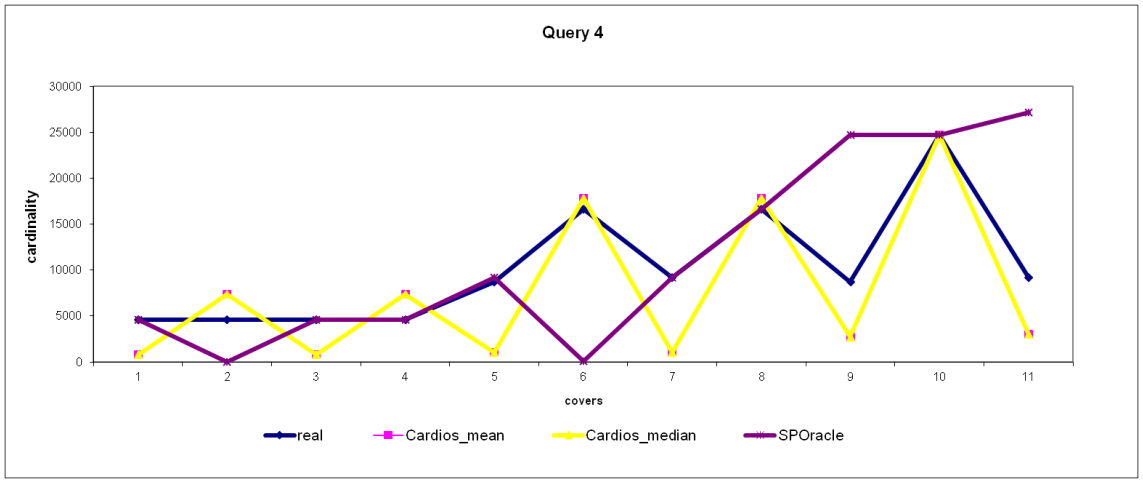
The high error estimates show that our models are far from being perfect. They also highlight that there is a wide spectrum of improvement in our estimation schema. In query1, the high estimation error is due to the inherent failure of mean or median aggregate functions to represent the extremes. Contrary, the join method estimates better such cases as it implements statistics directly from the materialized queries.

Table 3.18: Errors generated by CardiOS and SPOracle. Values indicate the average error when using the mean and median functions in CardiOS and join size estimation in SPOracle over all covers and residuals (BSBM and SP^{2}B).

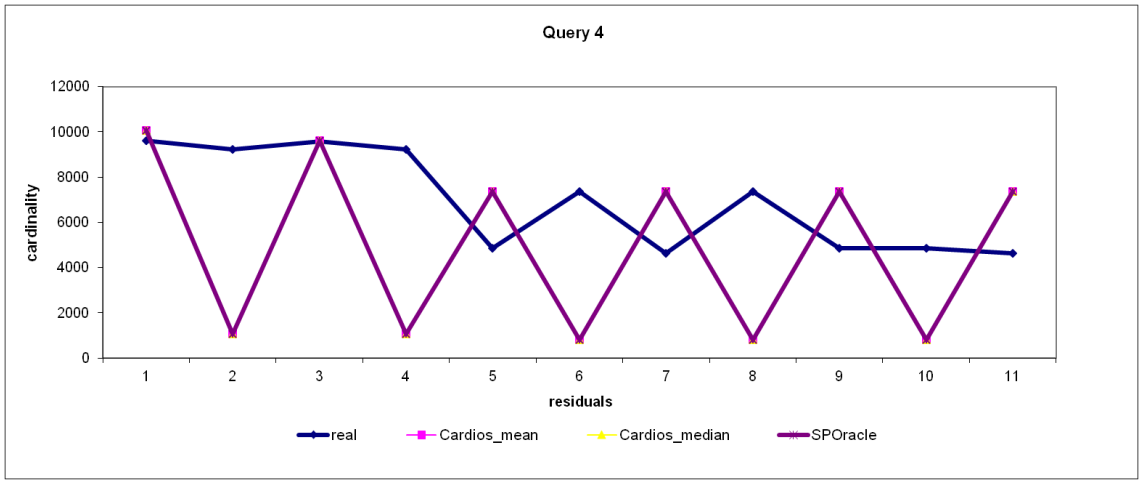
query		**CardiOS$_{\mathbf{mean}}$**	**CardiOS$_{\mathbf{median}}$**	**SPOracle**
$query_1$	*covers*	73.9 %	76.4 %	46.6 %
	residuals	0.0 %	0.0 %	0.0 %
$query_2$	*covers*	0.0 %	8.6 %	48.9 %
	residuals	0.0 %	1.5 %	0.0 %
$query_3$	*covers*	8.7 %	8.8 %	48.6 %
	residuals	9.3 %	8.3 %	10.2 %
$query_4$	*covers*	55.5 %	55.5 %	53.0 % %
	residuals	60.3 %	60.3 %	60.3 %
$query_5$	*covers*	59.0 %	59.0 %	15.0 % %
	residuals	30.5 %	30.5 %	30.5 %
$query_6$	*covers*	19.0 %	87.0 %	42.0 %
	residuals	0.0 %	0.0 %	0.0 %

Table 3.19: Average standard deviation of estimations generated using the models: $CardiOS_{mean}$, $CardiOS_{median}$, and *SPOracle* over all covers and residuals (BSBM and SP^{2}B).

query		**CardiOS$_{\mathbf{mean}}$**	**CardiOS$_{\mathbf{median}}$**	**SPOracle**
$query_1$	*covers*	1.0 %	1.5 %	49.8 %
	residuals	0.0 %	2.4 %	0.0 %
$query_2$	*covers*	0.0 %	4.9 %	52.3 %
	residuals	0.0 %	4.3 %	0.0 %
$query_3$	*covers*	10.0 %	10.1 %	51.3 %
	residuals	22.3 %	19.1 %	23.3 %
$query_4$	*covers*	34.1 %	34.1 %	77.9 %
	residuals	32.6 %	32.6 %	32.6 %
$query_5$	*covers*	40.5 %	40.5 %	20.7 %
	residuals	39.8 %	39.8 %	39.8 %
$query_6$	*covers*	33.0 %	39.0 %	46.0 %
	residuals	0.0 %	0.0 %	0.0 %



(a) Covers of query 4.



(b) Residuals of query 4.

Figure 3.8: Evaluation of estimation on covers in query4 over SP^{2}B.

For instance, the *rdf:type* predicate contained in query1 has a minimal fan at subject of 1 and a maximal fan at subject of 4, the mean fan is 1.08 and the median fan is 1. The estimations based on these statistics remarkably differ from the real cardinality. Only very few subjects actually have four predicate edges with *rdf:type*, but all of them are product nodes. Unfortunately, this is exactly what query1 focuses on. Therefore, the average error of estimation of mean and median function of CardiOS is nearly 74%. On the other hand, SPOracle benefits from the cases where its cover estimates equal the real cardinality. Even when some cover estimates have errors, the mean error for the set of covers decreases noticeably.

SPOracle provides an additional advantage. When the cover to analyze contains only one index, its estimated cardinality equals the frequency of that index, i.e., it has 100% accuracy. Contrary, the mean and median functions analyze the patterns regarding predicate–statistics and try to approximate its cardinality, which evidently, in those cases, does not achieve the same accuracy as with the join function.

On the other hand, SPOracle also highlights deficits. For instance, if varieties of the join attributes are high and the number of join attributes is larger than the number of participating indexes, estimations may be lower than the real cardinality of the pattern. Queries 4 and 5 also evidence high error rates. In these cases, the constant elements present in both queries influence our strategy and leads to underestimate the pattern cardinality.

From this error analysis we conclude that, in general, the use of the mean function in the CardiOS model leads to higher accuracy estimation of pattern cardinality than the use of the median function. For covered patterns, SPOracle is on average even more precise than CardiOS.

3.5.4 Accuracy of Weighted Models: Evaluating Different Weights

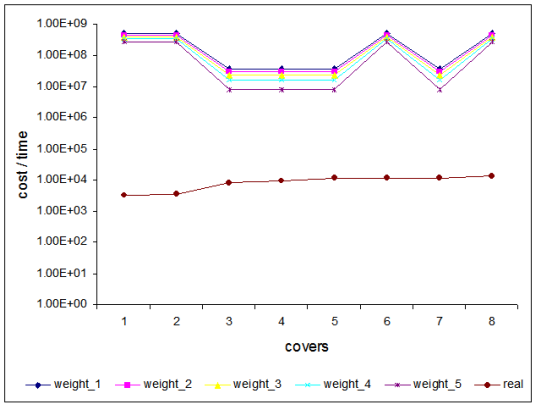
Figure 3.9 shows the evaluation of query1 using 10 different weights for both weighted models: $CardiOS_{mean}$ and SPOracle. Table 3.20 shows the different pairs of weights implemented in the models. Additional results for all test queries can be found in Appendix C Section 7.

Table 3.20: Evaluated cover and residual weights.

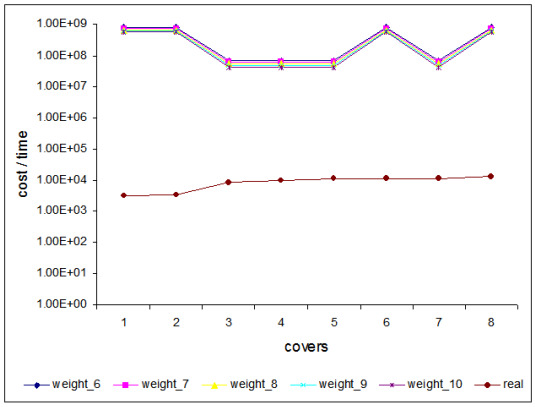
	cover	**residual**
$weight_1$	0.5	0.5
$weight_2$	0.4	0.6
$weight_3$	0.3	0.7
$weight_4$	0.2	0.8
$weight_5$	0.1	0.9
$weight_6$	0.9	0.1
$weight_7$	0.8	0.2
$weight_8$	0.7	0.3
$weight_9$	0.6	0.4
$weight_{10}$	0.55	0.45

The design of the charts allows to see the performance of the cost models using different weights. Series are given by solid lines. However, it does not mean that the plotted data is continuous but only simplifies to see the correlation between real and estimated values. Figure 3.9(a) and Figure 3.9(b) show the estimations for each cover of $query_1$ using all weight combinations over the SPOracle model. Similarly, Figure 3.9(c) and Figure 3.9(d) show the estimations for the same query using the $CardiOS_{mean}$ model. We selected these two models as they have shown higher estimation accuracy.

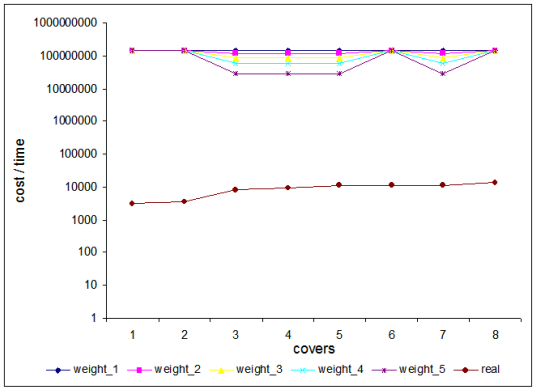
As expected, both models evidence higher correlation with the real execution time when the weights of the covers are lower than the weights of the residual ($cover_{weight} <$



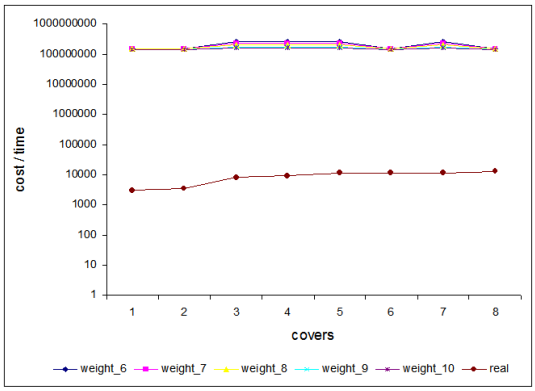
(a) SPOracle: Query 1 (BSBM)



(b) SPOracle: Query 1 (BSBM)



(c) $CardiOS_{mean}$: Query 1 (BSBM)



(d) $CardiOS_{mean}$: Query 1 (BSBM)

Figure 3.9: Comparison of estimated and real processing time for $query_1$ using SPOracle and $CardiOS_{mean}$. Estimations were done with 10 different weights over a 250k triples BSBM dataset.

$residual_{weight}$). However, there are some cases, e.g., Figure 3.9(d), where the correlation of estimated values for $query_1$ slightly improves in the opposite case ($cover_{weight} > residual_{weight}$) using the mean function on the CardiOS model. Table 3.21 shows estimates and real processing time for $query1$ using $cover_{weight} = 0.1$, $residual_{weight} = 0.9$ and $cover_{weight} = 0.9$, $residual_{weight} = 0.1$. Evidently, the order in which the covers should be executed is identified with more precision using $weight_6$. This fact highlights that the accuracy of the estimation depends not only on the statistics but also on the query structure. $Query_1$ is the only query that requests subjects with four predicate edges additional to the *rdf:type*. In the dataset, all nodes containing this predicate are product nodes. This case is better managed by the SPOracle model. Even when the estimations vary, mostly with larger values, they still show roughly the same correlation with the real execution time.

Table 3.21: Estimations and real processing time using $weight_5$ and $weight_6$ given in Table 3.20. Errors show that the query structure also influences the accuracy of the estimates.

cover	$\text{estimate}_{\text{w5}}$	$\text{estimate}_{\text{w6}}$	real time
$cover_1$	140,716,358	150,046,391	3,156
$cover_2$	140,716,358	150,046,391	3,489
$cover_3$	29,432,730	261,351,527	8,396
$cover_4$	29,432,730	261,351,527	9,579
$cover_5$	29,432,730	261,351,527	11,499
$cover_6$	140,716,358	150,046,391	11,505
$cover_7$	29,432,730	261,351,527	11,656
$cover_8$	140,716,358	150,046,391	13,295

3.5.5 Accuracy of Cost Models

In Section 3.5.2 and 3.5.3 we evaluated our estimations by separately measuring covered and residual patterns.

Now, we compare the results for the SycOM model with those of the CardiOS and the SPOracle model.

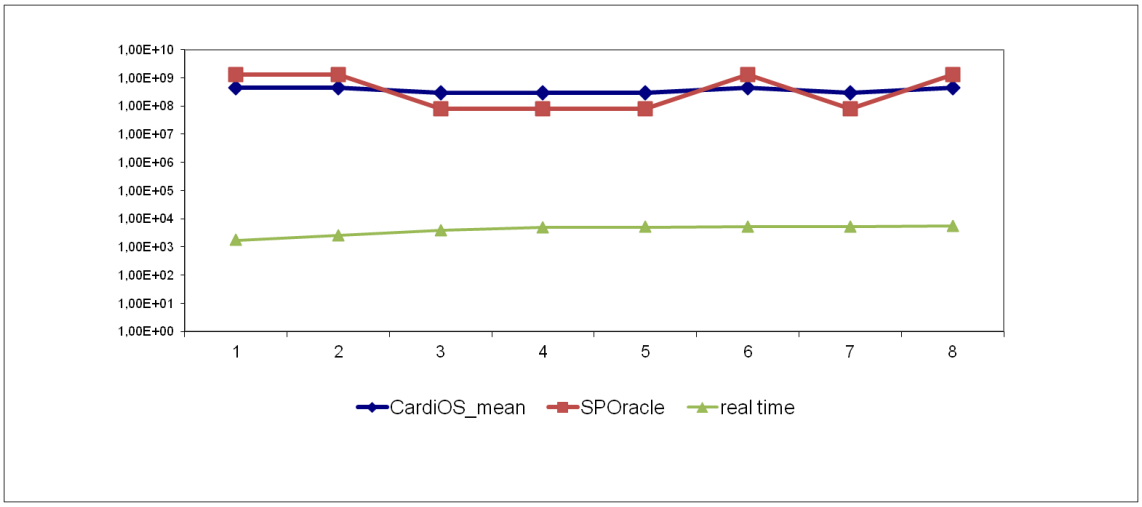
Our experiments are performed using two different configurations. First, we evaluate all execution plans for each test query and compare the overall processing time. Processing time is measured over 250K triples datasets from the Berlin and SP^2B benchmarks. Second, we compare the best processing time of each query (regarding each cost model) over four datasets ranging from 250 thousand to 10 million triples.

Figure 3.10 contains two charts showing the relation between estimated cost and real execution time. Each value represents the processing time the query engine consumes when implementing the i^{th} cover to answer the query. Based on the results introduced in Section 3.5.4, both models are evaluated using weights $wc = 0.33$ and $wr = 0.66$ to differentiate higher estimated residual cardinalities. Additional evaluation for all test queries is provided in Appendix C Section 8.

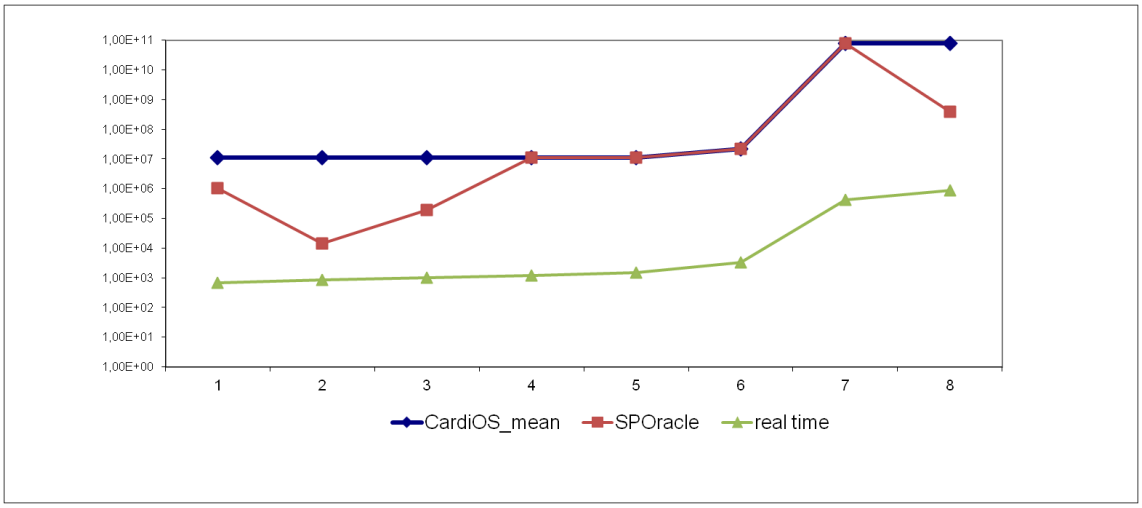
For each cover we display five measures. Three measures represent cardinality by using the $CardiOS_{mean}$, $CardiOS_{median}$, and SPOracle. The forth and fifth series describe SyCoM and real execution time. In the charts, the covers on the x-axis are sorted in increasing order according to their real running times. We ran each query/cover-combination seven times and disregard the highest and lowest running time from the set before computing the mean time. Notice that all data series are displayed in a logarithmic scale on the y-axis.

As can be seen from the charts, $CardiOS_{mean}$ and $CardiOS_{median}$ behaves very similar to SPOracle and mostly both succeed in predicting the correct order of the covers. On the other hand, SyCoM shows a more inaccurate overall estimation. The prediction of the order of the covers often differs from the real order, even when the selectivity estimation of each single cover's execution time (without residual part) correlates with its real selectivity

as seen previously in Figure 3.6.



(a) Query 1 (BSBM)



(b) Query 2 (BSBM)

Figure 3.10: Processing time vs estimates. For each test query all covers are estimated and executed over a 250K triples dataset (BSBM)

Figure 3.10(b) shows that the weighted mean and median function estimate the same values for some covers (e.g. covers 1 to 5). These cases occur when the estimated cardinalities for the covered and the residual part are equal. Notice also that the real running times for these covers range from 692 to 1492 milliseconds although the cost model using both functions (mean and median) rated them as equal. This shows that the cost model does not take into account all conditions to calculate a time–proportional cost. Nevertheless, the figures also show that the cost model predicts with some degree of accuracy the

real processing time of each cover.

SPOracle performs better in these particular cases. Even when the first cover is over–estimated the following four correlate with the real processing time. Clearly, with these estimations is not possible to select the *really fastest* cover. However, its degree of correlation is enough to identify a good cover, *nearly good enough as the best*, to process the query.

Figure 3.10 shows how our models evaluate each cover for a given test query. Now, in Figure 3.11 we show the best processing time the query engine returns when an optimal cover is selected. Covers are selected regarding estimated values predicted by the cost models implementing the mean, median, join and selectivity functions. The last column identifies the time required to execute the queries using a standard SPARQL processor (ARQ). In all charts y–axis is plotted in log–scale and the values are provided at the bottom. The evaluation is performed over four datasets containing 250K, 500K, 1M and 10M triples datasets respectively[6].

In accordance with our previous experiments, CardiOS using the mean function and SPOracle generally succeed in finding a better plan than ARQ. On the contrary, the SyCoM model does not. There are cases, e.g., query2, query3, query5 and query6 where the processing time of the selected cover only slightly improves or exceeds the standard processing time. Evidently, the residual part of the query, disregarded by this model, incurs an undesirable overhead.

Up to now, we have experimentally analyzed both models with all different functions. Clearly, the model SyCoM based on selectivity is significantly less accurate than CardiOS, using any of its statistical functions, and SPOracle. In the next section, we perform a comparison of the models using linear regressions and give their coefficients of determination. This analysis allows to compare the performance of the models quantitatively.

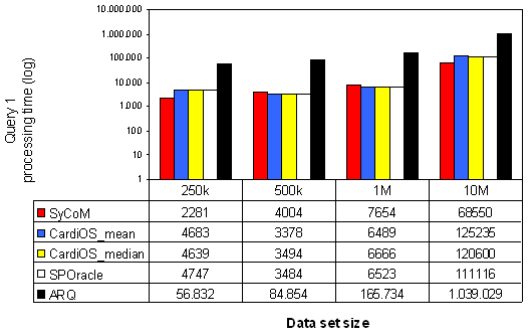
3.5.6 Comparing Cost Models

We now study whether CardiOS, or SPOracle predicted cost is proportional to the real execution time. We gather all available pairs of predicted cost and real execution time and display them in a scatter plot. The sampling pairs of predicted cost and real execution time are taken from all covers of all example queries over the Berlin and SP^2B benchmarks.

We generate two charts for the CardiOS and SPOracle models. First, we plot the raw values given by estimates and real processing time. The second chart shows the normalized version of these values rescaled to the square $((0,0),(5,5))$. For these models we compare both charts and show that, independently from the nature of the data, the accuracy of the prediction remains the same.

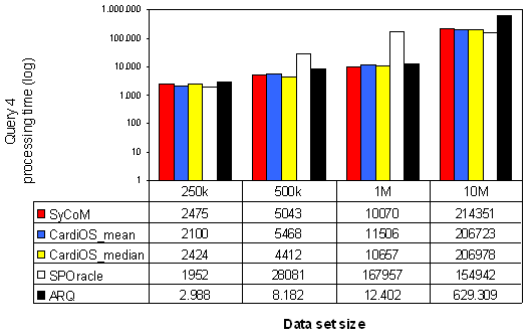
In the normalized version of the data, we expect the point for the "best" cover to be near the origin $(0,0)$ because a small predicted cost should indicate a small real execution time. Contrary, the "worst" cover should be situated somewhere near $(5,5)$ for each individual query. This fact would indicate that the highest predicted cost should coincide with the highest real execution time. In the best scenario all pairs of cost prediction and

[6]For simplicity $K = 1000, M = 1000000$



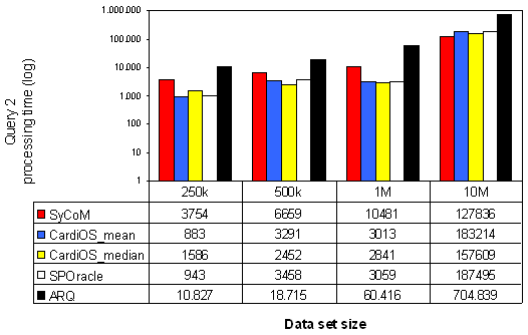
	250k	500k	1M	10M
SyCoM	2281	4004	7654	68550
CardiOS_mean	4683	3378	6489	125235
CardiOS_median	4639	3494	6666	120600
SPOracle	4747	3484	6523	111116
ARQ	56.832	84.854	165.734	1.039.029

(a) Query 1 (BSBM)



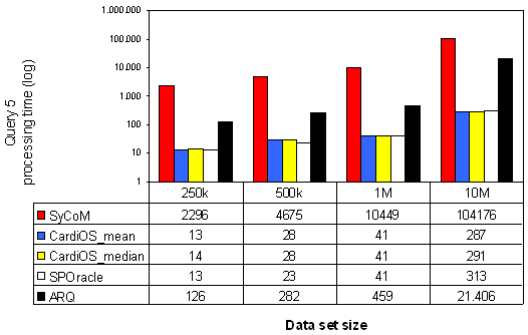
	250k	500k	1M	10M
SyCoM	2475	5043	10070	214351
CardiOS_mean	2100	5468	11506	206723
CardiOS_median	2424	4412	10657	206978
SPOracle	1952	28081	167957	154942
ARQ	2.988	8.182	12.402	629.309

(b) Query 4 (SP^2B)



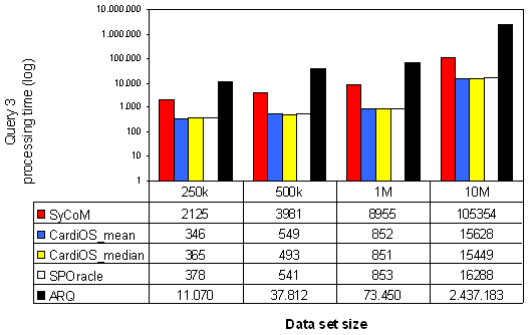
	250k	500k	1M	10M
SyCoM	3754	6659	10481	127836
CardiOS_mean	883	3291	3013	183214
CardiOS_median	1586	2452	2841	157609
SPOracle	943	3458	3059	187495
ARQ	10.827	18.715	60.416	704.839

(c) Query 2 (BSBM)



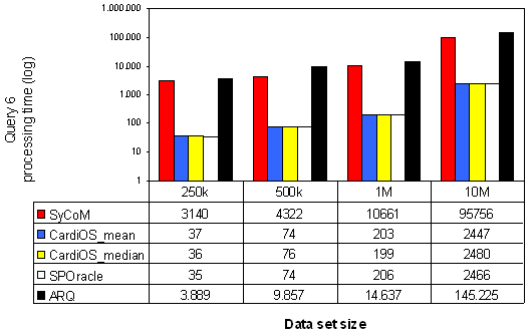
	250k	500k	1M	10M
SyCoM	2296	4675	10449	104176
CardiOS_mean	13	28	41	287
CardiOS_median	14	28	41	291
SPOracle	13	23	41	313
ARQ	126	282	459	21.406

(d) Query 5 (SP^2B)



	250k	500k	1M	10M
SyCoM	2125	3981	8955	105354
CardiOS_mean	346	549	852	15628
CardiOS_median	365	493	851	15449
SPOracle	378	541	853	16288
ARQ	11.070	37.812	73.450	2.437.183

(e) Query 3 (BSBM)



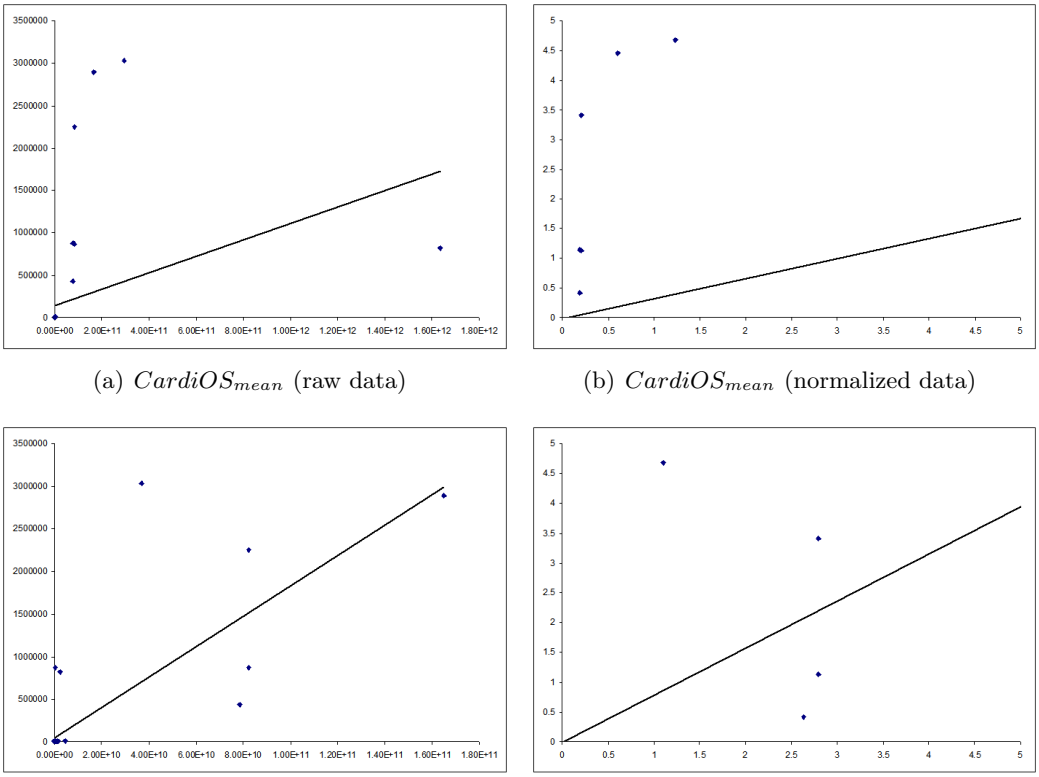
	250k	500k	1M	10M
SyCoM	3140	4322	10661	95756
CardiOS_mean	37	74	203	2447
CardiOS_median	36	76	199	2480
SPOracle	35	74	206	2466
ARQ	3.889	9.857	14.637	145.225

(f) Query 6 (SP^2B)

Figure 3.11: Processing time for test queries using BSBM and SP^2B. Each query was processed on four datasets using rewriting method MatView-and-ARQ (time in milliseconds).

real running time would be placed closed to the identity line. This would mean that the predictions are accurate.

For each set of data that relates the cost prediction to the real execution time, we calculate a linear fit function according to the least–squares–approach. Figure 3.12 shows the scatter plots of the raw and rescaled data along with their linear fit functions for CardiOS and SPOracle models.



(a) $CardiOS_{mean}$ (raw data)

(b) $CardiOS_{mean}$ (normalized data)

(c) SPOracle (raw data)

(d) SPOracle (normalized data)

Figure 3.12: Linear regressions for the CardiOs and SPOracle models.

The evidence provided by the visual analysis is conclusive. As stated before, the best fitting model must have a slope as close as possible to the identity line. The $CardiOS_{mean}$ model is good. However, the linear fit provided by SPOracle using the join size function matches this optimal case closer than any other model.

Table 3.22 and Table 3.23 list the linear function and coefficient of determination for CardiOS and SPOracle models. In both tables, the coefficient of determination, R^2, ranges from 0 to 1 to indicate how much confidence can be placed into the linear fit model to predict future pairs of cost model estimation and real execution time. In other words, the coefficient of determination is an indicator for the fitness of the cost model to predict the costs in a time–proportional fashion. The SPOracle model shows more confidence in its predictions than the CardiOS model. As can be seen in both tables, even when the linear

functions differ, the coefficient of determination for SPOracle is almost the same under both normalized and raw data.

Table 3.22: Linear regression properties on normalized data for CardiOS and SPOracle models.

cost model	**linear model**	$\mathbf{R^2}$
$CardiOS_{mean}$	$y = 0.3372x - 0.0203$	0.1132
$SPOracle$	$y = 0.7893x - 0.0071$	0.6223

Table 3.23: Linear regression properties on raw data for CardiOS and SPOracle models.

cost model	**linear model**	$\mathbf{R^2}$
$CardiOS_{mean}$	$y = 1 \cdot 10^{-6}x + 142556$	0.1112
$SPOracle$	$y = 2 \cdot 10^{-5}x + 50011$	0.62

3.6 Summary and Related Work

RDFMatView is an approach to use materialized SPARQL queries as indexes to improve the processing of other SPARQL queries. In this chapter, we enhanced this functionality by appropriate cost models based on statistical information. The goal was to develop a cost model that provides enough evidence to select a good execution plan for a given query. To achieve this goal, we developed three different cost models. The first cost model is based on an special form of index selectivity, the second model on cardinality estimation of query patterns and the third on a combination of join size and pattern cardinality estimations. We apply index statistics to estimate cover selectivity, predicate statistics to estimate the cardinality of triple patterns and a combination of both to estimate the complete query pattern.

Our initial model, SyCoM, sets the baseline. Results obtained using this model evidenced that our estimations correlate, in some degree of accuracy, with the cover execution time. However, they also showed that these estimations can be largely improved. Specifically, this model only considers the covered part of the query. Residual patterns are completely disregarded and lead to significantly inaccurate estimations of processing time.

We overcame this issue in the second model, CardiOS. We described a cost model that estimates the cardinality for the entire query pattern, i.e., covered and residual patterns. Additionally, the cost model has the ability to weight the residual part higher than the covered part of a pattern. This strategy influences the selection of execution plans towards those with a low residual pattern cardinality. We evaluated this model by using different aggregate functions, e.g. mean and median functions. This model is based on predicate

statistics gathered from the RDF dataset. Results showed that this model significantly outperformed the former model but could be still improved.

We enhanced the prediction accuracy by providing a third model, SPOracle, that combines join size estimation with pattern cardinality estimation. Statistical information is now taken from two sources, i.e., index and dataset metadata. Upon analysis of the results, this cost model have proved to be slightly more accurate than CardiOS. Although its estimations are not absolutely accurate, it succeeds in ordering the execution plans according to their real execution time. Its major advantage over its competitors lies on the combined use of statistical data, i.e., index properties and predicate statistics. Index properties are exact values from the predefined set of indexes computed at index processing time (e.g., frequency and index size).

Currently, as all cost models are designed for RDFMatView, they are restricted to estimating basic graph patterns. One line of future research might be to address this restriction and integrate the full set of SPARQL modifiers, e.g. FILTER, OPTIONAL and UNION into the cardinality estimation.

Related Work

Cost models are used to evaluate a set of execution alternatives for a given query. Based on this evaluation, the query engine might be able to select the fastest execution plan to answer the query. Cost models can differ from each other in the assumptions they take to provide their estimations and in the number of factors they take into account. However, they address the same objective disregarding the nature of their management systems, e.g.,relational, object–oriented or semantic [14].

Stocker et al. proposed in [86] a cost model to optimize basic graph patterns using selectivity estimation. They concentrate their efforts on providing static query optimization, i.e., reordering the join sequence of the triple patterns regarding their selectivity. Basically, the selectivity of a triple pattern is estimated by multiplying the selectivity of each single element of the triple pattern, i.e., subject, predicate and object (assuming statistical independence). However, this approach is restricted to in-memory models, i.e., datasets that can be fitted into memory. Evidently, the size of the statistical summary is a function of the dataset size. In large datasets, generating the summary will cause a massive bloating of statistical data preventing a proper performance of the system. Therefore, scalability is a clear disadvantage in this approach. Contrary, in all cost models presented in this chapter, the memory footprint is very small and scales according to the number of indexes or predicates, respectively.

In [61], Maduko et al. addresses cardinality estimation of RDF graph patterns. The authors implemented their approach by computing a semantic and structural summary. Essentially, this summary contains all subgraphs, up to a specified size, that may exist in the RDF dataset. To process the summary the authors assume an RDF schema for the dataset. All subgraphs of the dataset can be deduced from the schema graph and annotated with their exact cardinalities in the database. We share some similarities with this approach. In particular, we address the problem of estimating cardinality of patterns containing bound predicates. However, our approach does not rely on the use of a schema

graph and gathers statistical information by implementing heuristics to approximate cardinalities. With this strategy, we gain schema–independence.

Neumann and Weikum introduced in [68] an strategy to build statistics for all possible subgraphs in a schema-less dataset. They reduce the expressiveness of their subgraphs from arbitrary forms to chains and stars. Additional to these statistics, their approach uses a number of indexing capabilities to improve query processing. Actually, their approach heavily relies on its native indexing capabilities. Thus, the strategy is tightly coupled to its custom implementation and difficult to adapt to a different RDF semantic storage. In contrast, we propose an estimation strategy that can be seamlessly integrated into any SPARQL query processor.

In [82] Shironoshita et al. proposed an strategy for cardinality estimation built upon predicate–based statistics. The authors estimate the cardinality of the result set of a query over a given ontology applying a probability function. Similar to Maduko in [61], this strategy assumes a schema, denoted as data model, and is designed in the context of distributed systems.

4 Materialized View Selection: Selecting Materialized Views for RDF Data

The RDFMatView framework introduced in Chapter 2 requires a predefined set of materialized views (MV) to speed-up SPARQL query processing. In this chapter, we address the orthogonal problem of automatically suggesting an efficient set of MV for a given workload of SPARQL queries.

The index selection problem has been studied since the early 70's and its importance has been well recognized [25]. Several approaches have been proposed to solve the problem of MV selection, especially in relational databases and data warehouses [78, 3, 2, 35]. The basic principle of MV selection is to suggest a set of MV that optimizes the processing time of a given workload. The selection of this set is usually restricted by a given resource e.g., amount of space. Although well studied into relational databases, as far as we know, there are only very few approaches that study selection of MV for RDF data.

This chapter is structured as follows. In Section 4.1 the basic bricks of our approach are given. We define a workload as a set of SPARQL queries and show how to generate a candidate set of indexes suitable for that workload. Candidate indexes that can be applied for a specific query establish the basis for the final solution. In Section 4.2 we propose a cost model to evaluate each candidate index. The model estimates the potential impact of a set of candidate MVs over the entire workload. In Section 4.3 we introduce a comprehensive example for better understanding of our ideas. The implementation of our strategy is described in Section 4.4. We also describe an algorithm for evaluating the search space and for building a solution for the given workload. Section 4.5 provides an extensive evaluation. At the end of the chapter, we discuss our findings and provide related work. As throughout this thesis, in this chapter a materialized view is referred to as an index.

4.1 MV Selection Approach

In this section we introduce important notations. Definition 4.1 establishes the notion of a *workload* of SPARQL queries. Recall that, according to Definition 2.5, a SPARQL query consists of a set of triple patterns. We define a set of such queries as a workload in Definition 4.1.

Definition 4.1 (Workload of SPARQL queries) *Let $W = \{q_1, q_2, ..., q_n\}$ be a set of SPARQL queries and D an RDF dataset where $n \geq 0$. W is called a workload of SPARQL queries over D.* □

We also introduce the concept of *candidate index set*. We generate an initial search space of MV's by selecting all queries from W so as to see each query as a MV[1]. As mentioned by Goasduoe et al. in [34], materializing this solution is far from being optimal because its space consumption may be quite high, specially with large SPARQL workloads. However, we consider this approach only to generate a starting point. We extend this candidate index set by analyzing each query pattern and discovering all connected subgraphs of size $\geq k$. These subgraphs are treated as additional queries that could be materialized.

For each candidate index we estimate the gain it may achieve on each query using the cost model introduced further in Section 4.2. This process requires to apply a *query containment* algorithm to determine which pattern is contained in another query pattern (see Section 2.3). Additionally, it is necessary to estimate which indexes achieves more savings in time for W.

We refer to the resulting set of indexes as *candidate index set*. Definition 4.2 defines formally this concept.

Definition 4.2 (Candidate index set) *Let $W = \{q_1, q_2, ..., q_n\}$ be a workload of SPARQL queries and let $P(q_i)$ be the pattern of q_i, where $i = 1...n$. The set of candidate indexes for W is defined as the maximal set:*

$$V_c = \{mv_1, mv_2, mv_3, ..., mv_m\}$$

where:

- *$P(mv_j)$ is connected*
- *$P(mv_j) \subseteq P(q_i) \ \wedge \ |P(mv_j)| \geq k$*
- *$n \leq m$ and k is fixed.*

□

Table 4.1: An initial set of candidate indexes. At least one index is generated from each query of the workload.

	mv_1	mv_2	mv_3	**...**	mv_m
$query_1$	x	0	0	...	0
$query_2$	0	w	0	...	t
$query_3$	y	0	r	...	0
...	...	...	...	...	...
$query_n$	z	0	0		s

An index mv created from a query i sometimes is also eligible for a query q_j, if $P(mv) \subseteq P(q_j)$. We represent this information by computing a matrix mat, where rows represent the

[1]We assume that W contains no duplicates.

queries from the workload and columns the candidate indexes. Each value mat_{ij} represents a cost computed using the cost model introduced later in Section 4.2, Definition 4.4. An example is shown in Table 4.1. To decide which set of indexes from the candidate set are the most effective for the given workload, the system needs to evaluate all indexes and their influences on query processing. This decision should be made based on the expected reduction of time that an index achieves over the workload. We refer to these savings as *gain of an index* (see Definition 4.5). A value mat_{ij}, in the matrix described in Table 4.1, indicates which savings can be achieved by using index mv_j for $query_i$; if this value is 0, it means that the index is not eligible for the query. The total savings of an index can be computed by summing up all values in its column. Note that, our strategy heuristically estimates the savings of an index in the workload. These savings are unlikely to be reached in a real scenario. For instance, having $mv_1 \subseteq q$ and $mv_2 \subseteq q$, we sum savings of mv_1 and mv_2. However, only one index may be used during execution, depending on the evaluation performed by the query engine.

In Example 4.3 we describe the process of selecting indexes in a simple yet illustrative way. In a real scenario, a workload of queries usually contains a larger number of queries, which makes selecting an optimal set of indexes a difficult problem [88].

Example 4.3 (Selecting Indexes) *Assume a workload W consisting of three SPARQL queries q_1, q_2, and q_3 given below. q_1 asks for all universities, q_2 retrieves all professors working at each university, and q_3 returns the students studying at each university.*

q_1 :
```
SELECT * WHERE {
?university  rdf:type ub:University;
        ub:name ?uni_name.
}
```
q_2 :
```
SELECT * WHERE {
?uni rdf:type ub:University;
        ub:name ?uni_name.
?ub_AssistantProfessor ub:worksFor ?uni;
}
```
q_3 :
```
SELECT * WHERE {
?the_uni  rdf:type ub:University;
        ub:name ?uni_name.
?student ub:studyAt ?the_uni.
}
```

Clearly, we could materialize q_1 and reuse the result for the execution of all three queries, as q_1 is a sub-query of q_2 and q_3 (and, of course, also of q_1). On the other hand, q_1 also is very simple, and pre-computing it might not save much time for the entire workload. Suppose we were allowed to create only one index. In this case, we need to decide whether it is more advantageous to materialize q_1, gaining limited savings for all queries or, for

instance, materializing only q_3. This would only help to speed-up the query itself, but nevertheless could offer the highest total savings.

Evidently, in Example 4.3 all queries share triple patterns. Our idea is to discover those shared triple patterns such that they can be used as indexes to improve the processing time of the workload. However, to provide a solution for the selection problem, a cost model to evaluate each possible index and its influence over the workload is required.

4.2 Cost Model

To suggest an efficient set of indexes for a given workload we consider two facts:

- First, the estimated influence that the set could achieve on the workload.

  The influence on the workload can be seen as the achievement of runtime savings when using the set of indexes to process the given workload. To estimate these savings, in this section we propose a cost model based the cardinality of a pattern (see Chapter 3).

- Second, a given constraint of storage space (ρ).

  The storage constraint denotes the amount of disk space available to materialize the suggested indexes[2].

We use cardinality estimates, i.e., estimated number of triples, as indicator of the space that an index requires to be materialized in the system. Recall that indexes are selected offline, and therefore no estimations of their exact number of occurrences or processing time are available.

Our cost model disregards the cost of updating the selected indexes as RDF datasets and SPARQL workloads are typically read-only[3] and updates are performed only by using bulk transactions. Therein, all indexes must be recomputed from scratch.

We estimate the gain that an index offers to a query by comparing the costs it takes for executing the query with or without the index. We only look at the differences between different costs, which should roughly correlate with the savings in time (an evaluation of this approach is provided in Section 4.5).

Due to our limited knowledge about the queries, our model is built upon the definition of pattern cardinality used in the CardiOS cost model introduced in Chapter 3. Basically, the pattern cardinality is estimated by multiplying cardinalities of each single triple pattern. By implementing this function, we emphasize that executing patterns with larger cardinality requires larger processing time. Evidently, those queries that can be used to process a larger number of queries and that require larger processing time should be favored to be selected as indexes. Under this assumptions we denote the cost of a query q

[2] An estimation of which size of ρ is optimal for the workload is out of the scope of this work

[3] Currently, SPARQL does not provide an UPDATE transaction, although there are already proposals to add UPDATE to SPARQL (see [81]).

as $c(q)$, where $c(q) = estimate_{mean}(D, q)$ (see Definition 3.24 on page 75). $c(q)$ denotes the estimated number of results of q over a dataset D.

We use Definition 3.24 to estimate costs for both queries and candidate indexes. We refer to the cost of an index as the estimated time we could save when it is precomputed; thus, high costs are preferred. Having a high cost means that the index pattern covers query patterns containing a high estimated number of solutions in the dataset. In Definition 4.5 we introduce the function to estimate these costs.

As defined in Section 4.1, indexes of the candidate index set are generated from the given workload of SPARQL queries, taking each query pattern and its connected subgraphs of certain size as potential indexes. At processing time, a set of indexes is selected to be used in the execution of the query. However, it may happen that the selected set of indexes can process only a part of the query, i.e., there exists a residual part of the query pattern (r) that is not covered by the index pattern. This fact requires to estimate the processing time of using the index plus evaluating the residual part of the query.

We estimate a cost for processing r by looking at its cardinality. Thus, we apply again Definition 3.24 considering $q = r$. Using the query and residual costs, we turn now to define the cost of a query when using an index. Even though there exists a cost for retrieving the precomputed data, we assume the cost of an index $c(mv) = 0$. This assumption acknowledges the fact that retrieving one result of the covered pattern is faster than computing one result of the residual pattern. Covered patterns retrieve their solutions by joining the materialized queries stored in the underlying database system, i.e., the original data is not required. In opposite, residual patterns are executed against the RDF dataset by a SPARQL query engine. Executing the residual part requires a number of self joins on a triple table, which, depending on the size and complexity of the pattern as well as on the size of the dataset may generate costly processing time.

Definition 4.4 (Cost of a query when using an index) *Let q be a SPARQL query, mv an index and let r be the residual part of q when using mv, or q if mv is not eligible for q. The cost $c(q, mv)$ of executing q using mv is defined as follows:*

$$c(q, mv) = \begin{cases} c(q), & \textit{if } mv \textit{ not eligible for } q \\ 0, & \textit{if } |r| = 0 \\ c(r), & \textit{otherwise} \end{cases}$$

□

Based on the cost for executing a query with or without an index we can now define the gain of an index when used in a query.

Definition 4.5 (Gain of an index) *Let q be a query and mv an index. The gain of mv when used in q is defined as follows:*

$$gain_q(mv) = c(q) - c(q, mv)$$

Note that $gain_q(mv) = 0$ if mv is not eligible for q. □

We next define our optimal set of indexes given a workload as follows: The set, for which the sum of the gains of all its indexes is the highest under the given storage constraint. In Definition 4.6 we propose a model, *WorkQL*, to estimate the gain for a workload of SPARQL queries.

Definition 4.6 (WorkQL model: Gain for a SPARQL workload) *Let W be a workload of SPARQL queries and V_c a set of candidate indexes for W. The relative gain of an index $mv \in V_c$ over W is defined as follows:*

$$gain_W(mv) = \sum_{q \in W} gain_q(mv)$$

□

Additionally, we have:

Definition 4.7 (Optimal set of indexes) *Let V_c be a set of candidate indexes for a workload W. Let ρ be the maximum space that may be used to store indexes for the workload. A subset $S \subseteq V_c$ is called optimal for W iff the following holds:*

- $\sum_{mv \in S} gain_W(mv)$ *is maximal and*
- $\sum_{mv \in S} c(mv) \leq \rho$.

□

Note that, the costs of the materialized views are estimations of the space required to store them in the system. This implies that ρ cannot be seen as a strict border for the selected set. The constraint ρ intends to set up an intuitive threshold that avoids unlimited storage space.

4.3 Example

We provide an example to illustrate how we generate and evaluate the set of candidate indexes to suggest an efficient set, according to our cost model. We base this example on the workload of SPARQL queries described in Appendix A Section 1.

We create indexes from each query (as described in Section 4.1), such that for each query all connected components $\geq k$, where $k = 3$ are discovered. For instance, having a query q where $|P(q)| = n \ \wedge \ \forall t \in P(q) \ t_i \ \wedge \ t_j$ are connected , the number of possible combinations to get all connected components with size $\geq k$ would be $\sum_{i=k..n} \binom{n}{i}$ iff $n \geq k$. In our example, the number of candidate indexes generated from $query_1$ where $|P(query_1)| = 5$ is given by:

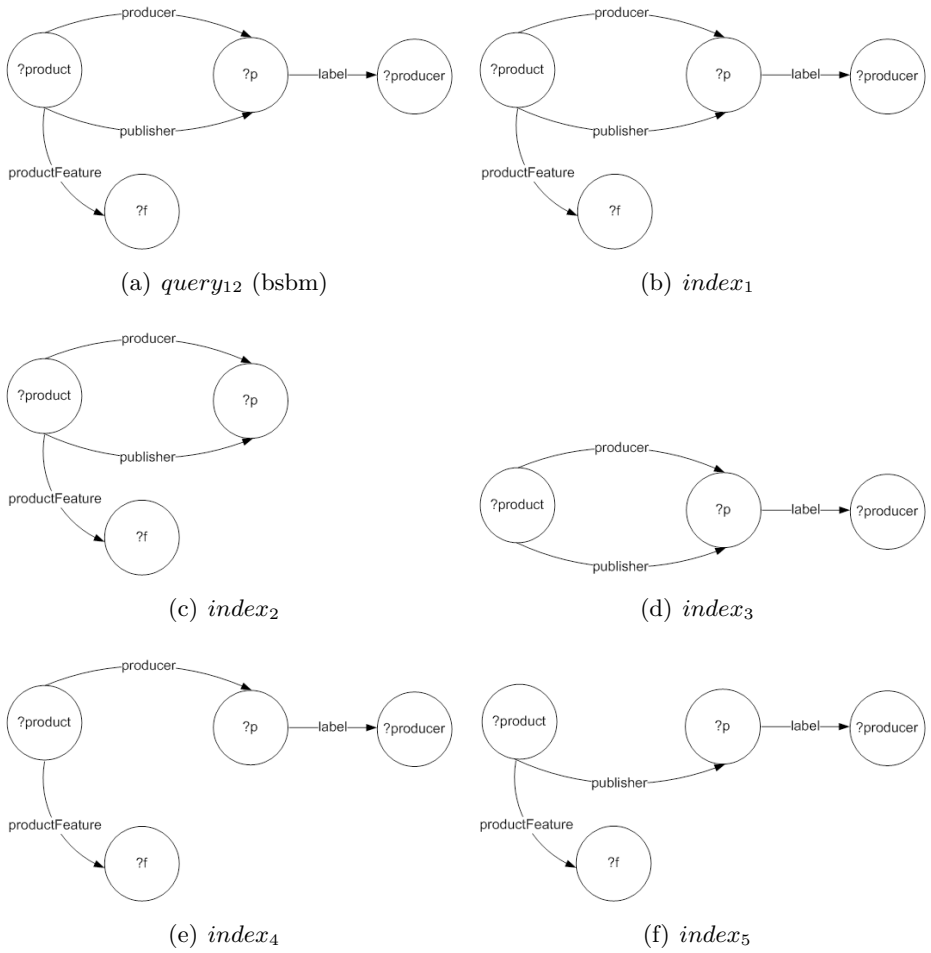
$$\sum_{i=3..5} \binom{5}{i} = 16$$

Note that $query_1$ denotes a star–shaped query where all triple patterns share a common variable, i.e., all triple patterns are connected with each other.

In general, assuming the worst case when all queries in W are star–shaped, the maximal number of indexes with size $\geq k$ that could be generated from W is given by:

$$\sum_{\forall q \in W} \sum_{i=3..n} \binom{n}{i} \tag{4.1}$$

Although Equation 4.1 denotes a hard border for the maximal number of indexes that can be generated from a workload W of SPARQL queries, in a real scenario the number of indexes may decrease according to the structure of each query pattern. Figure 4.1 illustrates all indexes that can be generated from $query_{12}$ of BSBM (see Appendix A Section 1).



(a) $query_{12}$ (bsbm) (b) $index_1$

(c) $index_2$ (d) $index_3$

(e) $index_4$ (f) $index_5$

Figure 4.1: Indexes generated from $query_{12}$ (BSBM).

Using the containment algorithm introduced in Chapter 2 Algorithm 2, we find mappings between index and query patterns. The process is performed iteratively until the overall influence of each index in the workload is found. The influence of an index is quantified by means of the WorkQL model (see Definition 4.6). In addition to the indexes described in Figure 4.1, Table 4.2 provides statistics of a set of indexes generated from $query_{10}$ and $query_14$ (BSBM). Each row describes the root query, index identifier, number of triple patterns, estimated cardinality, number of queries, and identifiers of those queries for which the index is eligible.

root query	Index	#patterns	estimated cardinality	#influenced queries	queries
$query_{10}$	$index_1$	3	666	3	$query_2$,$query_{10}$,$query_{12}$
$query_{10}$	$index_2$	4	16,381	2	$query_2$,$query_{10}$
$query_{10}$	$index_3$	3	16,382	2	$query_2$,$query_{10}$
$query_{10}$	$index_4$	3	16,381	2	$query_2$,$query_{10}$
$query_{10}$	$index_5$	3	16,381	2	$query_2$,$query_{10}$
$query_{14}$	$index_6$	3	6,659	2	$query_3$,$query_{14}$
$query_{14}$	$index_7$	5	6,659	2	$query_3$,$query_{14}$
$query_{14}$	$index_8$	4	6,659	2	$query_3$,$query_{14}$
$query_{14}$	$index_9$	3	6,660	2	$query_3$,$query_{14}$
$query_{14}$	$index_{10}$	4	6,659	2	$query_3$,$query_{14}$
$query_{14}$	$index_{11}$	3	6,659	3	$query_3$,$query_{14}$,$query_{15}$
$query_{14}$	$index_{12}$	4	6,660	2	$query_3$,$query_{14}$
$query_{14}$	$index_{13}$	3	6,660	2	$query_3$,$query_{14}$
$query_{14}$	$index_{14}$	3	6,660	2	$query_3$,$query_{14}$

Table 4.2: Index statistics over 250K triples dataset (BSBM).

As we notice in Table 4.2, $index_1$ can be used to process $query_2$, $query_{10}$ and $query_{12}$. Similarly, $index_{10}$ can be used to process $query_3$, $query_{14}$ and $query_{15}$. Most of the indexes can be used to process two queries, for instance $index_3$, $index_5$, and $index_12$. Although the number of queries that could be processed by an index is an important factor in the process of index selection, the suggested set of indexes is based on the estimated savings each index achieves at workload processing time. Additionally, the system verifies that the selected indexes do not deliberately violate the given constraint of storage (ρ).

4.4 Implementation

In this section, we describe the methodology to implement the materialized view selection problem upon the RDFMatView approach. Our implementation is based on the ARQ Jena API for Java 1.6 and PostgreSQL as relational backend system.

Overview

First, we describe the workflow of the system. It takes a workload of SPARQL queries and the amount of space as input. Each query pattern is analyzed and divided into its connected components. All connected component can be seen as a potential eligible indexes for the workload. Each eligible index is associated with a subset of queries from the workload, those that can be processed using the index in their execution plan. Using the cost model introduced in Section 4.2, the savings each query obtains when using that index

are estimated. Finally, we select those indexes that optimize the gains for the workload and satisfy the given constraint of the system. The workflow of this implementation is shown in Figure 4.2. A detailed description of each phase of the workflow is provided in the following section.

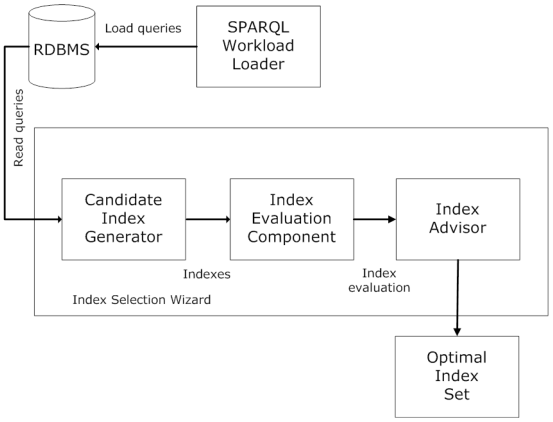


Figure 4.2: Workflow of an index selection approach. A workload of SPARQL queries is loaded into the system. The candidate index set is generated from the workload. Indexes are generated and evaluated using the cost model proposed in Section 4.2. Finally an index set is proposed.

Loading the Workload

The first step is to load a workload of SPARQL queries into the system. For each query we generate a unique identifier, which is mapped to its SPARQL definition. Additional statistical information from the given RDF dataset is stored as metadata in the underlying system such as frequency of subjects, predicate and objects, fan at subject and object, among others[4]. This information is requested by the system when evaluating the cost model.

Candidate Index Generation

The next step is to create a candidate index set. This process is done by analyzing each query pattern. Query patterns are grouped into all their connected components larger than k. These components represent the candidate index set. Algorithm 4 shows the analysis of query patterns to generate candidate indexes. Initially, the algorithm iterates over all queries of the workload. For each query, it groups those patterns that are connected. We iterate over these components and verify their number of triple patterns. If the number of patterns equals the constant k then we add the component to the set of candidate

[4]Definitions of these measures are introduced in Chapter 3

indexes. On the other case ($|P(q)|>k$), we apply a recursive algorithm to generate all possible combinations among the triples contained in the connected component (see Line 10). Details of this step are described in Algorithm 5. Essentially, it takes the component as a set of triple patterns and performs all possible combinations among them to generate all possible subsets. Notice that the subsets derived from the original component may contain components that are $\leq k$ and not necessarily connected. Therefore, in Line 11 and 12 we add operations to refine the set of indexes returned by Algorithm 5 to assure components $\geq k$ and to verify the connectedness of the refined set.

Note that to generate all subsets of connected components we implemented a brute force recursive algorithm. Its complexity is $O(2^n)$ where n denotes the size of the query pattern. Queries containing large query patterns may consume large processing time even if the pattern is not fully connected. An algorithm using pruning strategies for non–eligible indexes could fit more in the context of performance optimization.

Algorithm 4 getCandidates. Generating the candidate index set.

Given: W a Workload of SPARQL queries
Search: V_c a set of candidate indexes

```
 1: V_c ← [ ] ,  cc ← [ ] ,  nextElements ← [ ]
 2: for all queries ∈ W do
 3:    cc ← getConnectedComponents(query)
 4:    for all components ∈ cc do
 5:       if component.size ≥ k then
 6:          if component.size = k then
 7:             append(V_c, createIndex(component))
 8:          else
 9:             subsets ← [ ] ,  redset ← [ ] ,  allcc ← [ ]
10:             subsets ← getSubsets(component)
11:             redset ← refineSubsets(subsets)
12:             allcc ← verifyConnected(redset)
13:             for all component ∈ allcc do
14:                append(V_c, createIndex(component))
15:             end for
16:          end if
17:       end if
18:    end for
19: end for
20: return V_c
```

Index Evaluation

In this phase, each candidate index is evaluated using the WorkQL model (see Section 4.2) regarding the savings they provide when their influenced queries are processed. Therefore, a cost matrix *mat* is generated. This matrix is made up of queries as rows and indexes as columns. Each mat_{ij} value holds the gain of $query_i$ when using mv_j in its query execution plan. The total savings for each index is computed summing all savings from

Algorithm 5 getSubsets. Generating all components from a given pattern.

Given: *component* a component of triple patterns
Search: *allSubsets* the power set of c

```
1: if component.size = 0 then
2:    append(allSubsets, subset ← [ ]))
3: else
4:    redset ← [ ] ,  subsets ← [ ]
5:    append(redset, component)
6:    triple = redset[0]
7:    delete(redset, 0)
8:    subsets = getSubsets(redset)
9:    append(allSubsets, subsets)
10:   subsets = getSubsets(redset)
11:   for all subset ∈ subsets do
12:      append(subset, triple)
13:   end for
14:   append(allSubsets, subsets)
15: end if
16: return allSubsets
```

its corresponding column.

Selected Set of Indexes

Finding an optimal subset from all candidate indexes is not trivial. Choosing the optimal set under a space constraint is NP-Complete, as shown in [25]. However, there are fast approximation algorithms that have provable quality. Specifically, we use a greedy heuristic [27], which sorts the items (indexes) in decreasing order regarding estimated savings divided by space consumption. We then select indexes in decreasing order until the ρ parameter is reached. This algorithm guarantees that our solution is bounded with a value of at least $gain_W(J) \leq \frac{gain_W(V_c)}{2}$, where $J \subseteq V_c$ and $gain_W(V_c)$ is the maximal gain that can be achieved from the candidate index set using ρ storage space.

4.5 Evaluation and Results

As with the RDFMatView approach, all experiments were performed using two widely-accepted benchmarks: BSBM [10] and SP^{2}B [80]. The domain of the former is based on e-commerce use case information and the latter benchmark is regarding bibliographic information about the field of Computer Science and, particularly, databases (DBLP [58]). Using data generators provided in these benchmarks we create all datasets required to perform our experiments. We use the ARQ/Jena RDF Storage System (version 2.5.7) [5] and the RDFMatView approach [18] on Postgres 8.2 as framework. To measure the query processing time using RDFMatView we select the most efficient cover regarding the SPOracle model (see Definition 3.29).

4.5.1 Experiments

To evaluate the performance of our approach, we defined four different tests.

- First, we compare workload processing time (using a set of indexes) against standard processing time (without indexes) over 4 datasets with 250k, 500k, 1M, and 10M triples. The set of indexes is selected regarding a given constrain of storage space.
- Second, we evaluate the accuracy of our approach by creating three sets of indexes containing 5%, 10%, and 15% of storage space using our MV selection algorithm[5]. As in the previous test, these sets are used to process the entire workload and their processing time is compared to the workload processing time when using fifteen randomly generated index sets and to the standard processing time (without using indexes). All configurations are evaluated over an RDF dataset containing 500K triples.
- The third experiment compares the real versus the estimated storage consumption required to materialize the selected set of indexes using a constraint of 5% and 20% of storage space over all RDF datasets. We also describe how many indexes fit into the given storage constraint.
- Our last test compares the processing time of the workload with an increasing amount of space over a fixed dataset containing 1 Million triples.

From the set of queries provided by BSBM and SP^{2}B, we derived two workloads containing 18 and 15 queries according to the constraints of RDFMatView mentioned in Chapter 2. Using the algorithms introduced in Section 4.4 two sets of candidate indexes were generated. Each index pattern contains at least k triple patterns where $k = 3$.

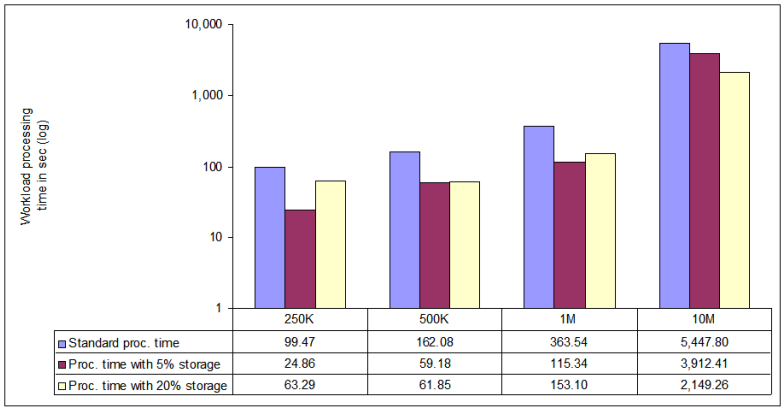
4.5.2 Results

Figure 4.3(a) and Figure 4.3(b) show the processing time for each workload over four datasets. The set of suggested indexes is generated considering 5% and 20% of the estimated storage required to materialize all queries of the workload. Evaluating a percentage of this total space we want to acknowledge that those selected indexes maximize the savings in time for the given workload.

In all scenarios, the workload processing time improves in comparison to standard query processing. However, Figure 4.3(a) shows that the reduction of the processing time achieved over small datasets is larger when the constraint of storage is set to 5%. This fact can be attributed mainly to the source query the indexes are generated from. For instance, the system suggests five indexes under a storage constraint of 5% over the 250K triples dataset. The number of indexes increases to ten when the storage constraint is set to 20%. In the first case, two indexes are derived from $query_1$. On the second case, four indexes are generated from the same query. Evidently, $query_1$ is the most costly query. Our method identifies this query and suggests those indexes that can be used to execute

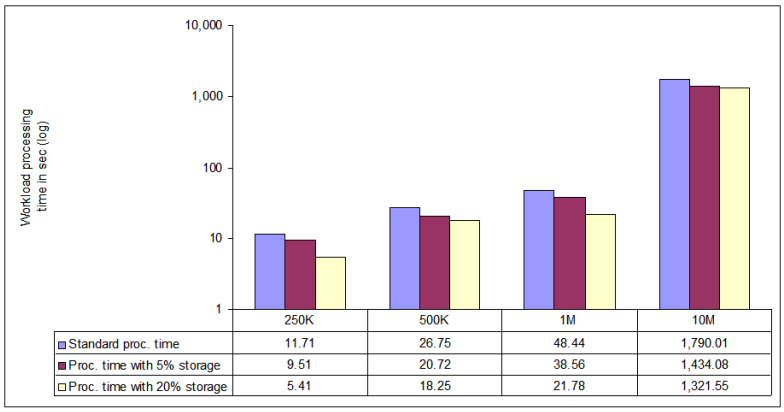
[5]Storage space denotes the sum of estimated costs required to materialize the complete SPARQL workload.

it regarding the given constraint. Nevertheless, in small datasets, using more indexes to execute a query may imply larger processing time because the number of required joins increases. The same indexes, over larger datasets, achieve better execution time.



	250K	500K	1M	10M
Standard proc. time	99.47	162.08	363.54	5,447.80
Proc. time with 5% storage	24.86	59.18	115.34	3,912.41
Proc. time with 20% storage	63.29	61.85	153.10	2,149.26

(a) Workload processing BSBM



	250K	500K	1M	10M
Standard proc. time	11.71	26.75	48.44	1,790.01
Proc. time with 5% storage	9.51	20.72	38.56	1,434.08
Proc. time with 20% storage	5.41	18.25	21.78	1,321.55

(b) Workload processing SP^{2}B

Figure 4.3: Workload processing time with and without time indexes. The graphics compare two sets of suggested indexes over BSBM and SP^{2}B datasets (250K to 10M triples). Indexes were selected according to 5% and 20% of the estimated storage required to materialize all candidate indexes. The graphic illustrates that for each dataset, the processing time for the given workload decreases when using the set of indexes suggested by our approach. Note that y-axis is plotted in log-scale (K = 1000, M = 1 Million triples).

Table 4.3 shows a list of indexes, selected under a constraint of 5% of storage. Similarly, Table 4.4 describes those queries that can be potentially processed using these indexes over a BSBM 250K dataset[6]. Note that, our selection strategy identifies those indexes suitable

[6]Storage values are given as estimated and real cardinalities of each index. Time is given in milliseconds.

to process costly queries, i.e., those queries with larger result sets. Each row of Table 4.3 describes the storage as estimated using our cost model and the real storage computed at index creation, the estimated savings per index and the number of queries, for which the index is eligible. Evidently, estimated storage correlates with the storage required to materialize the indexes. A detailed analysis of the correlation between estimated and real storage consumption is given later (Figure 4.5 and Figure 4.6).

Table 4.3: Set of selected indexes using 5% of storage

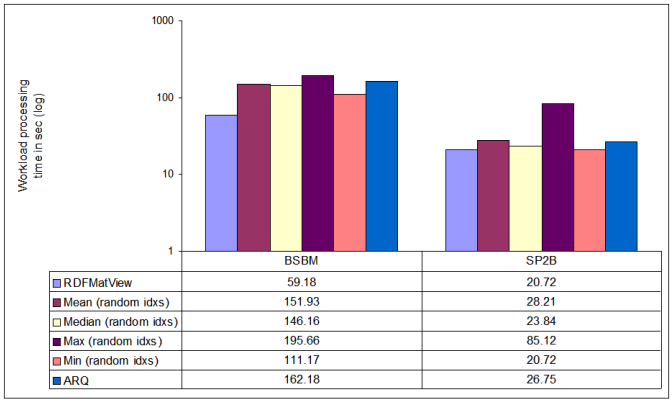
index	**estimated storage**	**real storage**	**estimated savings**	**#influenced queries**
$index_1$	17,750	65,528	437,982	2
$index_2$	721	2,664	33,656	1
$index_3$	9,316	11,502	9,316	1
$index_4$	1,332	1,310	4,665	1
$index_5$	666	666	666	2

Table 4.4: Queries processed using the selected indexes

query	**proc_time without idxs**	**proc_time with idxs**	**indexes used**
$query_1$	58,982	2,052	$index_1$, $index_3$
$query_2$	9,012	3,447	$index_5$
$query_3$	11,070	52	$index_2$
$query_8$	598	77	$index_1$
$query_12$	587	4	$index_5$

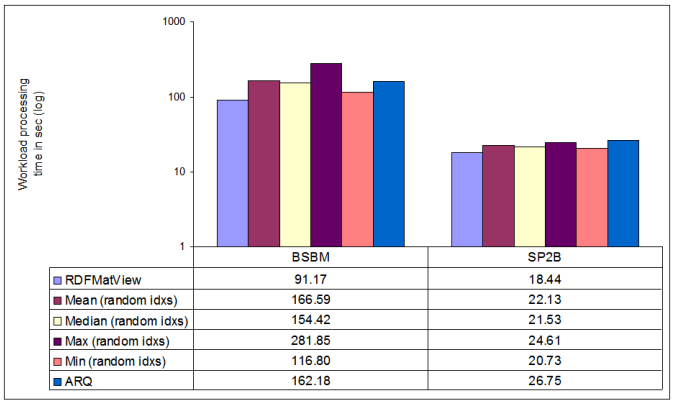
To verify the effectiveness of our approach and the accuracy of the selection algorithm, we compare the workload processing time when using selected and when using randomly generated sets of indexes. We select three sets of indexes (based on our cost model) using up to 5%, 10%, and 15% of storage space and fifteen sets of randomly generated indexes. The random selection is achieved by iteratively selecting indexes that fit within the storage constraint (also 5%, 10%, and 15% of storage space).

Figure 4.4 shows the results from evaluating the workloads using the RDFMatView system, plain ARQ (without indexes), and randomly selected indexes. Tests were performed over two datasets containing 500K triples. For the randomly selected indexes, different statistical measurements are reported (mean, median, minimum and maximum). Results show that our suggested sets of indexes are a good solution for the given workloads. Figure 4.4 shows that, in both domains, all index sets suggested by our system improve the standard query processing. The processing time using our selected indexes outperforms the minimal processing time using random index sets in 100% and 96% over the BSBM and SP^2B respectively. All figures show that the savings achieved over BSBM are larger



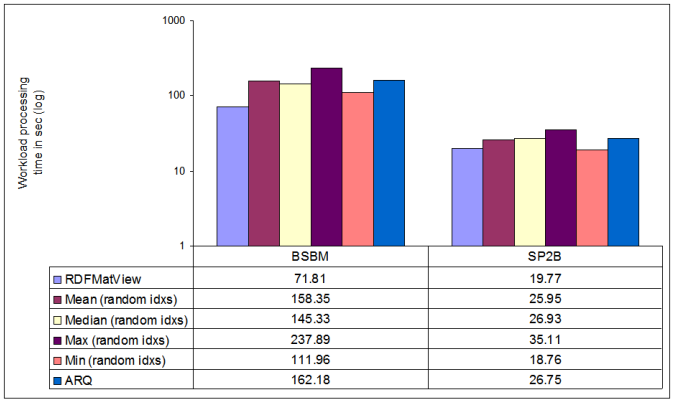
	BSBM	SP2B
RDFMatView	59.18	20.72
Mean (random idxs)	151.93	28.21
Median (random idxs)	146.16	23.84
Max (random idxs)	195.66	85.12
Min (random idxs)	111.17	20.72
ARQ	162.18	26.75

(a) Statistics 5% storage



	BSBM	SP2B
RDFMatView	91.17	18.44
Mean (random idxs)	166.59	22.13
Median (random idxs)	154.42	21.53
Max (random idxs)	281.85	24.61
Min (random idxs)	116.80	20.73
ARQ	162.18	26.75

(b) Statistics 10% storage



	BSBM	SP2B
RDFMatView	71.81	19.77
Mean (random idxs)	158.35	25.95
Median (random idxs)	145.33	26.93
Max (random idxs)	237.89	35.11
Min (random idxs)	111.96	18.76
ARQ	162.18	26.75

(c) Statistics 15% storage

Figure 4.4: Workload processing time using 5%,10%, and 15% of storage regarding the estimated storage for all candidate indexes over a 500K triples dataset. Results show that processing time decreases using our selected sets of indexes. Note that the processing time using our indexes is close to the minimum processing time when using randomly generated index sets.

than those obtained over SP^{2}B. Evidently, the number of indexes generated for the BSBM benchmark is quite large, and therefore, the probability to chose randomly an efficient set of indexes decreases. On the other hand, savings achieved over SP^{2}B are rather marginal for this dataset size and under the given constraints of storage. However, the workload processing time applying indexes suggested by our approach is very close to the minimal time achieved using randomly selected indexes.

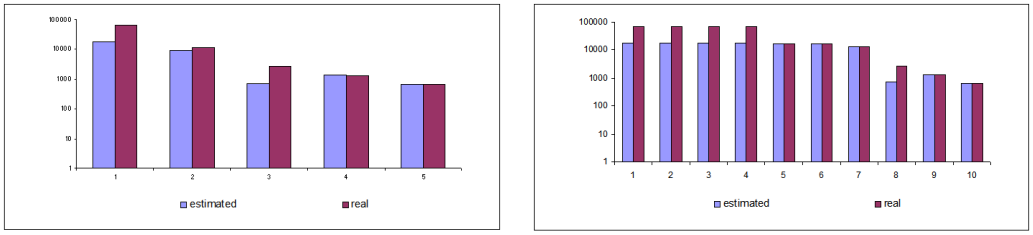
Table 4.5 shows how the selected set of indexes changes according to the given storage space over a 500K BSBM dataset. There are two indexes that are common to all sets, namely $index_1$ and $index_7$. These indexes are generated from $query_1$ and $query_3$, queries of the workload with larger real processing time. As can be seen in the table, our system succeeds in detecting those indexes eligible for speeding up their execution time.

Table 4.5: Selected indexes under a storage constraint of 5%, 10%, and 15%.

storage	$index_1$	$index_2$	$index_3$	$index_4$	$index_5$	$index_6$	$index_7$	$index_8$	$index_9$
5%	✓					✓	✓	✓	
10%	✓		✓				✓		✓
15%	✓	✓	✓	✓	✓	✓	✓	✓	✓

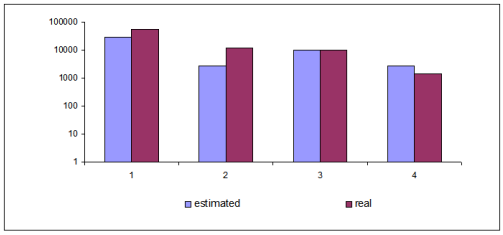
Figure 4.5 and Figure 4.6 show estimated vs. real storage consumption for each index selection over the benchmarks. Each pair of values is sorted by real storage in decreasing order to facilitate the correlation. Y-axis is plotted in log-scale. Notice that storage is represented by number of triples. Under this assumption, we first analyze each index and estimate its cardinality and secondly, for those selected indexes, we compute the real number of materialized results. Although the number of materialized results denotes an exact computation of the cardinality of one index, its physical storage consumption may significantly vary regarding this value. The visual analysis of Figure 4.5 and Figure 4.6 is conclusive. Estimated storage is far from being perfect regarding real storage. However, apart from some exceptions, estimated and real storage in general correlate, and therefore, provide a suitable storage consumption estimation.

Finally, Figure 4.7 shows that having an increasing number of indexes available for the query processor improves the workload processing time. In RDFMatView, we differentiate two cases, in which indexes can be used to process a query. The first case is when the query can be completely processed using indexes, i.e., the selected set of indexes covers the entire query pattern. The second case is when the query can be processed only partially, i.e., there exists a residual part of the query pattern that need to be processed without indexes. Evidently, the first case is the most profitable since no additional processing is required (except for join operations required among selected indexes). In the second case, a residual pattern need to be matched against the original RDF dataset and joined with the partial results of the covered part of the query. Although this process may demand larger execution time, it is usually more efficient than executing the original pattern. According to these argumentation, the more indexes are selected, the more queries may be covered, and therefore, the lower the workload processing time.

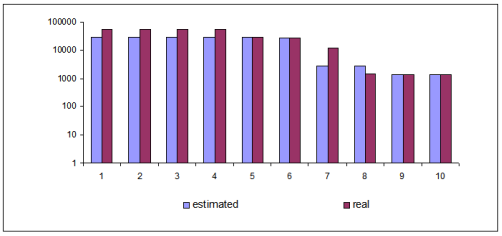


(a) Selection 5% (250K)

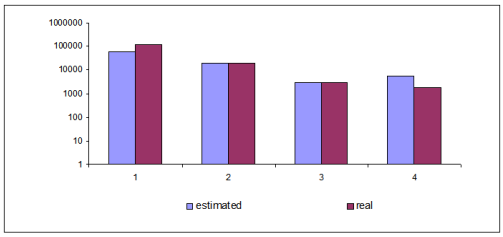
(b) Selection 20% (250K)



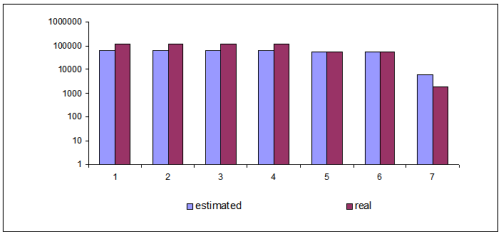
(c) Selection 5% (500K)



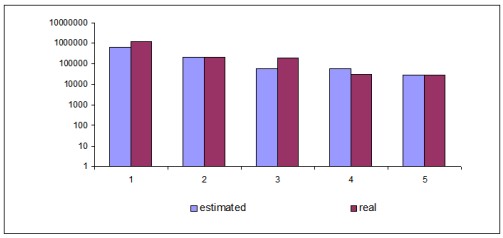
(d) Selection 20% (500K)



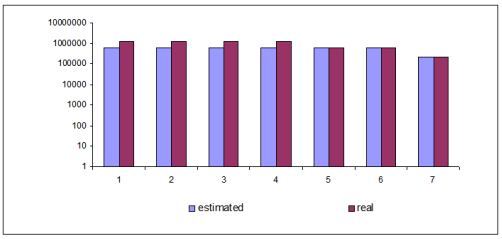
(e) Selection 5% (500K)



(f) Selection 20% (500K)

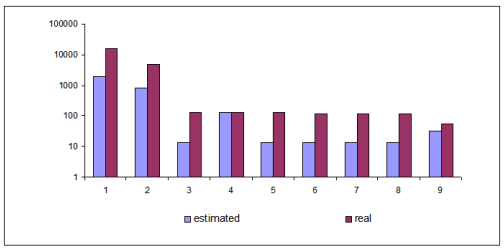


(g) Selection 5% (10M)

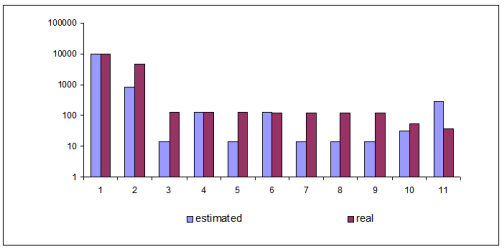


(h) Selection 20% (10M)

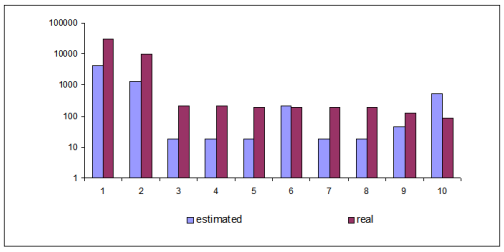
Figure 4.5: Estimated vs real storage consumption over 250K, 500K, 1M and 10M triples datasets (BSBM).



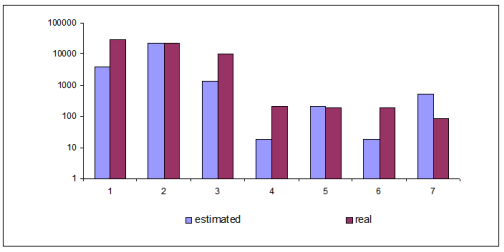
(a) Selection 5% (250K)



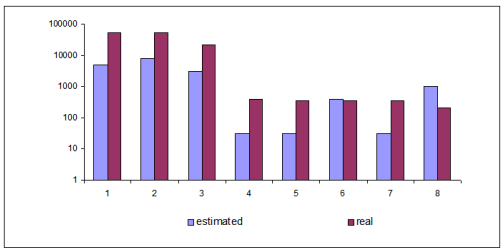
(b) Selection 20% (250K)



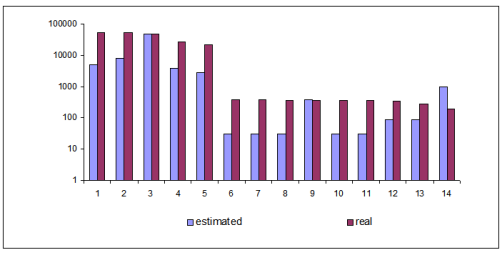
(c) Selection 5% (500K)



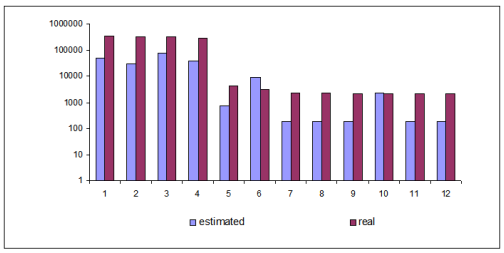
(d) Selection 20% (500K)



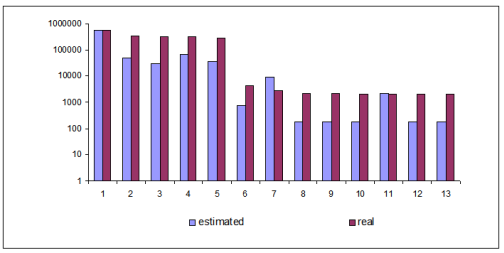
(e) Selection 5% (500K)



(f) Selection 20% (500K)

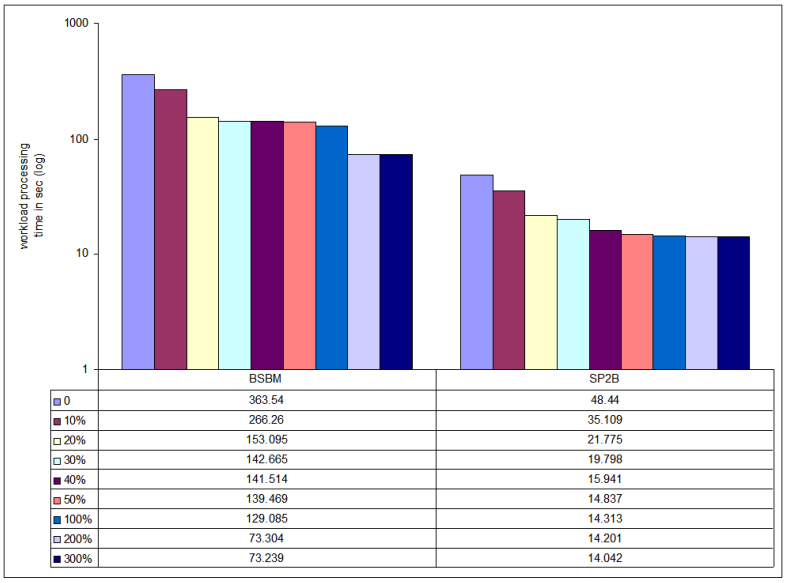


(g) Selection 5% (10M)



(h) Selection 20% (10M)

Figure 4.6: Estimated vs real storage consumption over 250K, 500K, 1M and 10M triples datasets (SP^2B).



	BSBM	SP2B
0	363.54	48.44
10%	266.26	35.109
20%	153.095	21.775
30%	142.665	19.798
40%	141.514	15.941
50%	139.469	14.837
100%	129.085	14.313
200%	73.304	14.201
300%	73.239	14.042

Figure 4.7: Workload processing time using standard query processing (without indexes) and using from 0 to 300% storage space over datasets with 1 Million triples (BSBM and SP^{2}B). The graphic shows how the processing time decreases as the amount of storage space available for the query processor increases.

4.6 Summary and Related Work

We introduced a system that suggests an efficient set of indexes for a given workload of SPARQL queries. Candidate indexes are created by discovering all connected component of each query pattern with size $\geq k$. The application of this set of indexes in the query processing helps to improve the entire workload processing time.

To this end, we provide WorkQL, a cost model to estimate savings in processing time for a given workload. We generate and evaluate all candidate indexes and estimates the processing time regarding the cardinality of query pattern as described in Chapter 3. The estimated value aims to capture the influence each index has in the workload during processing time.

Up to now, our approach is restricted to the type of queries required by RDFMatView. We currently work with a constraint on storage space, however, we want to emphasize that implementing any other constraint in the system is straightforward if additional index information is provided, e.g., number of results, processing time or number of indexes. One line of future research might be to address optimization of the index selection process analyzing not only the connected graph components but also generating potential overlapping indexes[7] regarding these new generated patterns. We believe that such an approach should improve the quality of the suggested indexes as RDFMatView uses indexes with

[7] indexes with triple patterns in common

overlapping properties aiming to cover as many patterns from the query as possible.

Related Work

In relational database systems the problem of index selection has been continuously addressed since the early 70's [25]. In [16], Caprara et al. propose a practical solution for the index selection problem based on a workload of SQL queries using heuristics. They use a branch and bound algorithm to find a reduced set of candidate indexes. Chaudhuri et al. propose an index selection tool in [23]. They divide the problem into three basic stages: First, generation of a set of candidate indexes and selection of those, which are more promising based on the query syntax and estimated cost. Second, optimization algorithms to evaluate sets of candidate indexes, and finally, iterative generation of more complex indexes from the simpler "good" alternatives.

An enhancement of this approach is proposed in [3]. Additional to automated index selection, the authors improve the system by allowing to choose materialized views for relational databases. Contrary to previous works, (e.g. [16]), this approach considers the combination of classical index structures, such as B+ trees, and materialized views to define an optimal physical design for a database system.

In [41], Gupta et al. addressed the view–selection problem applied to data warehouse design. This approach introduces different heuristics to derive a suitable set of MV's from a given workload of relational queries build upon a set of source tables. The authors apply the concept of directed acyclic graphs (DAG's) to represent the queries of the workload. Each subgraph resulting from merging these expressions is considered as a potential view for materialization. The idea is to discover alternative ways to evaluate a given query from the base relations and other views. The authors also introduce a cost model to estimate the maintenance cost of the views. From these costs and considering the space consumption, Gupta et al. provide different greedy algorithms to evaluate the candidate views and help to select those views that minimize the workload processing time. In contrast to Gupta et al., we derive our potential views by generating all possible connected components (with size $\geq k$) from each single query pattern. We analyze each index at the time, estimate its cost and let the generation of execution plans to the query engine. The authors also use the advantage that in relational systems the set of source tables is known in advance. This fact adds valuable information about the data and allows to evaluate potential execution plans. Contrary, we apply our approach over RDF datasets. RDF can be seen as schema–free databases and therefore, information about the data is given only by the query structure.

In [59], Liang et al. investigate the view selection problem specifically under the maintenance time constraint. Similar to [41], the authors emphasize the importance of selecting an efficient set of materialized views in data warehouse design. On the other hand, Liang et al. argue that disk space should no longer be considered as the major limiting factor for the view selection problem as the ratio between price and disk capacity drops continuously. Instead, the authors evaluate the selection of views by constraining the highly time–consuming processes required to keep materialized views up–to–date. The view selection is achieved by using two heuristics. The first algorithm consists of two phases that separately evaluate processing time and select by maintenance cost. The second heuristic

describes an algorithm that simultaneously evaluates both factors, processing and maintenance time. Although selecting views under constraints of storage and maintenance time seems to be very similar, they are significantly different. In the former, adding new views always increments the disk consumption, whereas in the latter adding new views does not necessarily imply higher maintenance time. This work is thus orthogonal to our because we work under a constraint of storage space. Additionally, SPARQL does not support UPDATE transactions (there are currently proposals to add this functionality to SPARQL (see [81])). Updates are performed by massively loading data. Thus, in our approach adding new data implies that all indexes must be recomputed from scratch.

Theodoratos et al. propose in [88] a method for data warehouse design using views. From a given set of relational queries, their process generates a set of materialized views that satisfy all input queries. Contrary to [59], the authors formulate their approach under a constraint of disk space. However, they emphasize that the set of views should minimize the overall query evaluation and the maintenance time. Views are generated applying a set of predefined transformation to the input query definitions. The selected views should be materialized and allow a complete rewriting of the input queries. Contrary to this approach, our set of suggested views does not necessary satisfy all input queries. Additionally, selected views may only be suitable to partially execute a query, therefore, additional operations are required to join the partial results of the covered and uncovered part of the query.

In [34] Goasdoue et al. touch on the topic of materialized view selection while looking for an approach to improve query processing over RDF datasets. This approach comes close to our approach but cannot be directly compared because of its dependence on an RDF schema. Moreover, this approach is constrained to queries that can be completely executed by using the selected set of materialized views. The authors begin by creating candidate view sets based on existing proposals for relational data and queries. As the complexity for this problem is very high, the authors present a set of heuristics to make it tractable. Specifically they adapt the approach of [88] for view selection in data warehouses. Basically, they approach the view selection problem as a search problem and apply a set of transformations to the workload until an optimal set of views is reached.

Similar to [34], we propose an automated selection of a set of indexes in the form of materialized views. However, contrary to [34], our set of indexes may be used to partially cover a query and interacts with an standard SPARQL query engine. Our indexing strategy fully exploit the RDF graph-structure. We are not indexing single attributes or triples, but fractions of queries that occur frequently in a given workload. Therefore, our approach suggests a set of native RDF/SPARQL indexes whose concepts are viable for all possible implementations of RDF stores.

5 Summary and Outlook

During the last years, the *WWW* is undergoing a metamorphosis from containing human–readable information to machine–processable data. This has been driven thanks to the *Semantic Web* community and its efforts to define a structured representation for concepts and their relationships on the web. The basic brick of the Semantic Web is the development of *RDF*, a data model that represents resources by using the notion of triples consisting of subjects, predicates and objects *(s,p,o)*.

The level of success achieved with this new paradigm can be perceived by the increasing number of RDF datasets published on the web. The adoption of the Semantic Web by the scientific and industrial community has motivated the development of interesting approaches addressing topics such as semantic storage, indexing, query languages, query processing, linked data, provenance, trustworthiness, semantic annotation, and so on [47].

In this thesis, we addressed indexing and query optimization by proposing an approach for integrating materialized views into SPARQL query processing. We believe that due to the abundance of RDF information on the web, extracting and analyzing semantic information efficiently has become a necessity and a challenging task.

The use of materialized views in query processing has been proved to be a scalable strategy to speed–up queries in relational database systems [6]. Therefore, inspired on this practice, we propose a novel approach that exploits the natural graph–representation of the RDF data and makes persistent those graphs, which upon analysis, heavily influences the processing of a given workload of SPARQL queries.

5.1 Summary of the thesis

We contribute to the Semantic Web community by formally defining a logical and physical framework to integrate materialized views into SPARQL query processing. This contribution, however, is just the major concept that comprises a set of specific tasks required to accomplish the objective of using materialized views in SPARQL query processing. We identified these tasks as follows:

- Optimization of SPARQL query processing by using materialized SPARQL queries as indexes
- Rewriting of SPARQL queries to include materialized views in their execution plan
- Selection of an optimal execution plan (using materialized views) from a set of generated alternatives to efficiently answer a SPARQL query

- Development of a cost–based approach for selecting an optimal set of materialized views from a given workload of SPARQL queries
- Modelling of statistical cost functions to evaluate generated executions plans, such that an optimal selection could be suggested regarding their estimated value.

In this thesis, we provide solutions for all these tasks. After motivating our dissertation in the first chapter, in Chapter 2 we present RDFMatView, an approach which allows the use of materialized views into SPARQL query processing. Furthermore, we provide specific algorithms, which integrated into a SPARQL query processor, make the persistent stored graphs available for execution of queries.

We devoted Chapter 3 to describe and develop cost models to evaluate different plans. Initially, we introduced SyCoM, a selectivity–based cost model. By taking very simple assumptions, this model try to estimate the query processing time looking only at the covered query patterns. Results showed significant improvements in query processing, but also highlighted new directions for further research. Therefore, we introduced an enhanced model CardiOS, regarding pattern cardinality. Basically, this cost model uses predicate–based statistics to evaluate selectivity of triple patterns. Contrary to the former model, CardiOS considers covered and uncovered patterns from the given query, which improves the accuracy of the estimated value. We observed remarkable improvement in query processing regarding the former model. However, CardiOS is based on predicate-statistics only and disregards statistics stemming from the set of predefined indexes. Index statistics are exact values that are computed from the indexes at index processing time. We combined index and predicate statistics and proposed a third model: SPOracle. In this model, we pursue a more accurate cardinality pattern estimation by dividing the query pattern into a covered and a residual part. We applied index statistics to the covered part and predicate statistics to the residual part of the query pattern. This model outperforms both former models in terms of improving the evaluation of the execution plans as shown by using queries from the BSBM and the SP^2B Benchmarks.

In Chapter 4 we describe our solution for the orthogonal problem of selecting an optimal set of materialized views given a workload of queries (materialized view selection problem). Initially, in RDFMatView we assumed a given set of materialized views previously selected based on their frequency over a given workload and a dataset. Here, we propose a strategy to enrich the RDFMatView framework that automatically – and solely regarding a given workload – suggests an optimal or close to an optimal set of views to materialize according to a workload–savings–based cost model. We integrate this functionality into RDFMatView and round off the complete functionality of the framework (except materialized views updates, this topic is not considered in this thesis).

5.2 Future research directions

The Semantic Web is facing every day new challenges in a wide range of fields. In this thesis we address a narrow but important part of its topics. Our results contribute by bringing new methodologies to scale–up these semantic queries but they also leave open

questions to solve and set the basis for further development. Therefore, we highlight a list of possible research directions related with RDF indexing and query processing.

In our framework we use relational database technology to base the implementation of the RDFMatView approach. This decision gave us the possibility to rely on the matureness of relational systems. There are, however, nowadays storages engines designed specifically for graph–modelled data built upon native non–relational storage systems [91, 68]. Usually, these systems also provide a wide range of indexing strategies, which enhances the retrieval capabilities of their underlying data. Implementing the use of materialized views on these systems and, in combination with their native indexing capabilities may achieve important savings in query processing time.

We also restricted our RDFMatView implementation to basic graph patterns. Extensions for patterns using SPARQL modifiers such as *FILTER, OPTIONAL, UNION, GROUP BY* would also improve the functionality of this approach.

Chapter 3 introduces three different statistical cost models to evaluate graph patterns. Besides the advantages that they provide in the selection of a good execution plan, there are aspects that can be improved. For instance, dealing with cycles in query patterns. In our current prototype, cycles are detected to avoid infinite loops in such a way that all edges are traversed. Our estimation strategy simply multiplies the fans for each predicate (regarding all different roots), disregarding the special characteristic of cycles. Further investigation may exhibit opportunities for better overall cardinality estimation.

The evaluation of our approaches was performed using two well-know SPARQL bechmarks, namely the Berlin and the SPARQL Performance Benchmark (BSBM and SP^2B respectively). Both provide data generators, which allowed us to create datasets with different sizes by configuring the scale factor. However, due to the synthetic nature of the generated data, datasets usually have identical value distributions, which in real–world datasets may be significantly different. Therefore, further evaluation using real–world data could provide valuable information and evidence tasks that can be refined to improve the accuracy of our approach.

Bibliography

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management using Vertical Partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 411–422, 2007.

[2] Serge Abiteboul and Oliver M. Duschka. Complexity of Answering Queries Using Materialized Views. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 254–263, 1998.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 496–505, 2000.

[4] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. Clustering-Based Materialized View Selection in Data Warehouses. In *In ADBIS 2006, volume 4152 of LNCS*, pages 81–95, 2006.

[5] ARQJena. ARQ - A SPARQL Processor for Jena, 2010. URL `http://jena.sourceforge.net/ARQ/`.

[6] Inderpaal Singh Ashish Gupta. *Materialized Views: Techniques, Implementations, and Applications*. The MIT Press, 1999.

[7] Tim Berners-Lee and Jeffrey Jaffe. W3C Home Page, 2010. URL `http://www.w3.org/`.

[8] Tim Berners-lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *In Proceedings of the 3rd International Semantic Web User Interaction Workshop*, 2006.

[9] Bio2RDF. http://bio2rdf.org/, 2009.

[10] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *International Journal On Semantic Web and Information Systems - Special Issue on Scalability and Performance of Semantic Web Systems, 2009*, 2009.

[11] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. *SIGMOD Rec.*, pages 61–71, 1986.

[12] Dan Brickely and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, 2004. URL `http://www.w3.org/TR/rdf-schema/`.

[13] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *International Semantic Web Conference*, pages 54–68. 2002.

[14] Nicolas Bruno and Surajit Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 263–274, 2002.

[15] O. Peter Buneman and Eric K. Clemons. Efficiently Monitoring Relational Databases. *ACM Trans. Database Syst.*, 4:368–382, 1979.

[16] Alberto Caprara, Matteo Fischetti, and Dario Maio. Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design. *IEEE Transactions on Knowledge and Data Engineering*, 7:955–967, 1995.

[17] Roger Castillo and Ulf Leser. Selecting Materialized Views for RDF Data. In *Semantic Web Information Management Workshop (SWIM 2010)*, 2010.

[18] Roger Castillo, Ulf Leser, and Christian Rothe. RDFMatView: Indexing RDF Data for SPARQL Queries. Technical Report 234, Humboldt Universitaet zu Berlin, 2010.

[19] Roger Castillo, Christian Rothe, and Ulf Leser. Indexing RDF Data using Materialized SPARQL Queries. In *Scalable Semantic Web Knowledge Base Systems Workshop (SSWS 2010)*, 2010.

[20] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 577–589, 1991.

[21] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of Computing*, pages 77–90, 1977.

[22] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries with Materialized Views. *Data Engineering, International Conference on*, page 190, 1995.

[23] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 146–155, 1997.

[24] Artem Chebotko, Shiyong Lu, Hasan M. Jamil, and Farshad Fotouhi. Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical report, Department of Computer Science, Wayne State University, 2006.

[25] Douglas Comer. The Difficulty of Optimum Index Selection. *ACM Trans. Database Syst.*, 3:440–445, 1978.

[26] Thomas M. Connolly, Carolyn E. Begg, and Anne D. Strachan. *Database Systems: A Practical Approach to Design, Implementation and Management*. 1996.

[27] George Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1998.

[28] UniProt: RDF Dataset. http://dev.isb-sib.ch/projects/uniprot-rdf/.

[29] Chandrashekhar A. Dhote and M. S. ALi. Materialized View Selection in Data Warehousing. *Information Technology: New Generations, Third International Conference on*, pages 843–847, 2007.

[30] George H.L. Fletcher and Peter W. Beck. Scalable Indexing of RDF Graphs for Efficient Join Processing. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 1513–1516, 2009.

[31] Johann Christoph Freytag. The Basic Principles of Query Optimization in Relational Database Management Systems. In *IFIP Congress*, pages 801–807, 1989.

[32] Marc Friedman, Alon Levy, and Todd Millstein. Navigational Plans for Data Integration. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1999.

[33] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 2000.

[34] Francois Goasdoue, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. Materialized View-Based Processing of RDF Queries. In *BDA 2010*, 2010.

[35] Jonathan Goldstein and PerAke Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 331–342, 2001.

[36] Jinghua Groppe, Sven Groppe, Sebastian Ebers, and Volker Linnemann. Efficient Processing of SPARQL Joins in Memory by Dynamically Restricting Triple Patterns. In *Proceedings of the 24th ACM Symposium on Applied Computing (ACM SAC 2009)*, pages 1231–1238, 2009.

[37] Sven Groppe, Jinghua Groppe, and Volker Linnemann. Using an Index of Precomputed Joins in order to speed up SPARQL Processing. In *Proceedings 9th International Conference on Enterprise Information Systems (ICEIS 2007 (1), Volume DISI)*, pages 13–20, 2007.

[38] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. In *International Conference on Enterprise Information Systems*, 2005.

[39] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *SIGMOD Rec.*, 22:157–166, 1993.

[40] Himanshu Gupta. Selection of Views to Materialize in a Data Warehouse. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, pages 98–112, 1997.

[41] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *Database Theory - ICDT 97*, volume 1186 of *Lecture Notes in Computer Science*, pages 98–112. 1997.

[42] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 311–322, 1995.

[43] Alon Y. Halevy. Answering Queries Using Views: A Survey. *The VLDB Journal*, 10: 270–294, 2001.

[44] Stephen Harris. SPARQL Query Processing with Conventional Relational Database Systems. In *International Workshop on Scalable Semantic Web Knowledge Base System*, 2005.

[45] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009.

[46] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *LA-WEB '05: Proceedings of the Third Latin American Web Congress*, page 71, 2005.

[47] Olaf Hartig. Querying Trust in RDF Data with tSPARQL. volume 5554, pages 5–20. 2009.

[48] Ralf Heese, Ulf Leser, Bastian Quilitz, and Christian Rothe. Index Support for SPARQL. *European Semantic Web Conference, Innsbruck, Austria*, 2007.

[49] Ivan Herman. The Semantic Web home page, 2010. URL `http://www.w3.org/2001/sw/`.

[50] Alice Hertel, Jeen Broekstra, and Heiner Stuckenschmidt. RDF Storage and Retrieval Systems. In *Handbook on Ontologies*, pages 489–508, 2009.

[51] Yannis Ioannidis. The History of Histograms (abridged). In *Proc. of VLDB Conference*, 2003.

[52] Yannis E. Ioannidis. Query Optimization. *ACM Comput. Surv.*, 28:121–123, 1996.

[53] Clement Jonquet, Paea LePendu, Sean M. Falconer, Adrien Coulet, Natalya F. Noy, Mark A. Musen, and Nigam H. Shah. NCBO Resource Index: Ontology-Based Search and Mining of Biomedical Resources. In *2010 Semantic Web Challenge*, 2010.

[54] Shaye Koenig and Robert Paige. A Transformational Framework for the Automatic Control of Derived Data. In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, pages 306–318, 1981.

[55] Rosana S. G. Lanzelotte and Patrick Valduriez. Extending the Search Strategy in a Query Optimizer. In *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 363–373, 1991.

[56] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, 2002.

[57] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering Queries Using Views (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104, 1995.

[58] M. Ley. DBLP Database. URL `http://www.informatik.uni-trier.de/~ley/db`.

[59] Weifa Liang, Hui Wang, and Maria E. Orlowska. Materialized view selection under the maintenance time constraint. *Data & Knowledge Engineering*, 37:203 – 216, 2001.

[60] Baolin Liu and Bo Hu. An Evaluation of RDF Storage Systems for Large Data Applications. *Semantics, Knowledge and Grid, International Conference on*, 0:59, 2005.

[61] Angela Maduko, Kemafor Anyanwu, Amit Sheth, and Paul Schliekelman. Estimating the Cardinality of RDF Graph Patterns. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1233–1234, 2007.

[62] Frank Manola and Eric Miller. RDF Primer, February 2004. W3C Recommendation.

[63] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. An Indexing Scheme for RDF and RDF Schema based on Suffix Arrays. In *Proceedings of SWDB 2003*, pages 151–168, 2003.

[64] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language, 2004. URL `http://www.w3.org/TR/owl-features/`.

[65] Hagen Moebius. A Cost Model for Optimizing SPARQL Queries with RDFMatView. Master's thesis, Humboldt University of Berlin, 2010.

[66] MySQL. http://dev.mysql.com/, 2010.

[67] Thomas Neumann and Guido Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *International Conference on Data Engineering 2011*, 2011.

[68] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style Engine for RDF. *Proc. VLDB Endow.*, 1:647–659, 2008.

[69] Ralph Heese Olaf Hartig. The SPARQL Query Graph Model for Query Optimization. In *ESWC2007*, 2007.

[70] Brian McBride Patrick Hayes. RDF Semantics, February 2004. W3C Recommendation.

[71] Axel Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th international conference on World Wide Web*, pages 787–796, 2007.

[72] Postgres. http://www.postgresql.org/, 2011.

[73] W3C SWEO Community Project. Linking Open Data on the Semantic Web., 2009. URL `http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData/`.

[74] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF, April 2008. URL `http://www.w3.org/TR/rdf-sparql-query/`. W3C Recommendation.

[75] Greg Riccardi. *Principles of Database Systems with Internet and Java Applications*. Addison Wesley, 2001.

[76] Christian Rothe. Indexierung von RDF-Daten für SPARQL-Anfragen. Diplomarbeit, Humboldt-Universität zu Berlin, Juli 2007.

[77] Nicholas Roussopoulos. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *ACM Trans. Database Syst.*, 16:535–563, 1991.

[78] Nick Roussopoulos. Materialized Views and Data Warehouses. *SIGMOD Rec.*, 27: 21–26, 1998.

[79] Simon Schenk. A SPARQL Semantics Based on Datalog. In *KI 2007: Advances in Artificial Intelligence*, volume 4667, pages 160–174. 2007.

[80] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. *International Conference on Data Engineering*, pages 222–233, 2009.

[81] Andy Seaborne, Geetha Manjunath, Chris Bizer, John Breslin, Souripriya Das, Ian Davis, Steve Harris, Kingsley Idehen, Olivier Corby, Kjetil Kjernsmo, and Benjamin Nowack. SPARQL Update: A Language For Updating RDF Graphs, July 2008. W3C Member Submission.

[82] E. Patrick Shironoshita, Michael T. Ryan, and Mansur R. Kabuka. Cardinality Estimation for the Optimization of Queries on Ontologies. *SIGMOD Rec.*, 36:13–18, 2007.

[83] Oded Shmueli and Alon Itai. Maintenance of Views. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 240–255, 1984.

[84] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2006.

[85] Nicholas Gibbins Stephen Harris. 3Store: Efficient Bulk RDF Storage. In *1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, 2003.

[86] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 595–604, 2008.

[87] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. On Rules, Procedure, Caching and Views in Data Base Systems. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 281–290, 1990.

[88] Dimitri Theodoratos, Spyros Ligoudistianos, and Timos Sellis. View Selection for Designing the Global Data Warehouse. *Data & Knowledge Engineering*, 39:219 – 240, 2001.

[89] Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. GRIN: A Graph Based RDF Index. In *AAAI*, pages 1465–1470, 2007.

[90] María-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *The Semantic Web: Research and Applications*, volume 6088, pages 228–242, 2010.

[91] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proc. VLDB Endow.*, 1:1008–1019, 2008.

[92] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. First International Workshop on Semantic Web and Databases*, 2003.

[93] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph Indexing Based on Discriminative Frequent Structure Analysis. *ACM Trans. Database Syst.*, 30:960–993, 2005.

[94] H. Z. Yang and PerAke Larson. Query Transformation for PSJ-Queries. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 245–254, 1987.

[95] Clement T. Yu and Weiyi Meng. *Principles of Database Query Processing for Advanced Applications*. 1998.

[96] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient Exploitation of Similar Subexpressions for Query Processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 533–544, 2007.

Appendix A Queries and Indexes (BSBM)

1 Berlin SPARQL Benchmark (BSBM)

All queries and indices of BSBM use the following namespaces:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX bsbm-inst:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX rev:<http://purl.org/stuff/rev#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
```

Listing 21: Common namespace prefixes.

1.1 Queries

```
SELECT * WHERE {
    ?product rdfs:label ?label .
    ?product a ?ProductType .
    ?product bsbm:productFeature ?ProductFeature1 .
    ?product bsbm:productFeature ?ProductFeature2 .
    ?product bsbm:productPropertyNumeric1 ?value1 .
}
```

Listing 22: Query 1

```
SELECT * WHERE {
    ?product rdfs:label ?label .
    ?product rdfs:comment ?comment .
    ?product bsbm:producer ?p .
    ?p rdfs:label ?producer .
    ?product dc:publisher ?p .
    ?product bsbm:productFeature ?f .
    ?f rdfs:label ?productFeature .
    ?product bsbm:productPropertyTextual1 ?propertyTextual1 .
    ?product bsbm:productPropertyTextual2 ?propertyTextual2 .
    ?product bsbm:productPropertyTextual3 ?propertyTextual3 .
    ?product bsbm:productPropertyNumeric1 ?propertyNumeric1 .
    ?product bsbm:productPropertyNumeric2 ?propertyNumeric2 .
 }
```

Listing 23: Query 2

```
SELECT * WHERE {
    ?product rdfs:label ?productLabel .
    ?offer bsbm:product ?product .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .
    ?vendor rdfs:label ?vendorTitle .
    ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
    ?offer dc:publisher ?vendor .
    ?offer bsbm:validTo ?date .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
    ?review dc:title ?revTitle .
    ?review bsbm:rating1 ?rating1 .
}
```

Listing 24: Query 3

```
SELECT * WHERE {
    ?vendorURI rdfs:label ?vendorname .
    ?vendorURI foaf:homepage ?vendorhomepage .
}
```

Listing 25: Query 4

```
SELECT * WHERE {
    ?offer bsbm:vendor ?vendorURI .
    ?offer bsbm:offerWebpage ?offerURL .
    ?vendorURI rdfs:label ?vendorname .
    ?vendorURI foaf:homepage ?vendorhomepage .
}
```

Listing 26: Query 5

```
SELECT * WHERE {
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?review bsbm:rating1 ?rating1 .
}
```

Listing 27: Query 6

```
SELECT * WHERE {
    ?offer bsbm:product ?product .
    ?offer bsbm:vendor ?vendor .
    ?offer dc:publisher ?vendor .
    ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
    ?offer bsbm:deliveryDays ?deliveryDays .
    ?offer bsbm:price ?price .
    ?offer bsbm:validTo ?date .
}
```

Listing 28: Query 7

```
SELECT * WHERE {
    ?offer2 bsbm:offerWebpage ?offerURL2 .
    ?offer2 bsbm:price ?price .
    ?offer2 bsbm:deliveryDays ?deliveryDays .
}
```

Listing 29: Query 8

```
SELECT * WHERE {
    ?product bsbm:productPropertyNumeric1 ?value1 .
}
```

Listing 30: Query 9

```
SELECT  * WHERE {
    ?product  rdfs:label      ?label ;
    rdf:type        ?ProductType;
    bsbm:productFeature ?ProductFeature1 .
}
```

Listing 31: Query 10

```
SELECT * WHERE {
    ?product a ?ProductType .
    ?product bsbm:productFeature ?ProductFeature1 .
}
```

Listing 32: Query 11

```
SELECT * WHERE {
    ?product bsbm:producer ?p .
    ?p rdfs:label ?producer .
    ?product dc:publisher ?p .
    ?product bsbm:productFeature ?f .
 }
```

Listing 33: Query 12

```
SELECT * WHERE {
    ?product bsbm:productFeature ?f .
    ?f rdfs:label ?productFeature .
 }
```

Listing 34: Query 13

```
SELECT * WHERE {
    ?product bsbm:producer ?p .
    ?offer bsbm:product ?product .
    ?offer bsbm:vendor ?vendorURI .
}
```

Listing 35: Query 14

```
SELECT * WHERE {
    ?product rdfs:label ?productLabel .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
    ?review dc:title ?revTitle .
}
```

Listing 36: Query 15

```
SELECT * WHERE {
    ?review bsbm:reviewFor ?product .
    ?review dc:title ?title .
    ?review rev:text ?text .
}
```

Listing 37: Query 16

```
SELECT * WHERE {
    ?review bsbm:reviewFor ?product .
    ?review bsbm:rating1 ?rating1 .
}
```

Listing 38: Query 17

```
SELECT * WHERE {
    ?offer bsbm:product ?productURI .
    ?productURI rdfs:label ?productlabel .
    ?offer bsbm:vendor ?vendorURI .
    ?offer bsbm:price ?price .
}
```

Listing 39: Query 18

1.2 Indexes

```
SELECT * WHERE {
    ?product rdfs:label ?label ;
    rdf:type ?ProductType ;
    bsbm:productFeature ?ProductFeature1 .
}
```

Listing 40: Index 1

```
SELECT * WHERE {
    ?product a ?ProductType .
    ?product bsbm:productFeature ?ProductFeature1 .
}
```

Listing 41: Index 2

```
SELECT * WHERE {
    ?product bsbm:producer ?p .
    ?p rdfs:label ?producer .
    ?product dc:publisher ?p .
    ?product bsbm:productFeature ?f .
 }
```

Listing 42: Index 3

```
SELECT * WHERE {
    ?product bsbm:productFeature ?f .
    ?f rdfs:label ?productFeature .
 }
```

Listing 43: Index 4

```
SELECT * WHERE {
    ?product rdfs:label ?label .
    ?product rdfs:comment ?comment .
    ?product bsbm:producer ?p .
    ?p rdfs:label ?producer .
    ?product dc:publisher ?p .
    ?product bsbm:productPropertyTextual1 ?propertyTextual1 .
    ?product bsbm:productPropertyNumeric1 ?propertyNumeric1 .
}
```

Listing 44: Index 5

```
SELECT * WHERE {
    ?product rdfs:label ?productLabel .
    ?offer bsbm:product ?product .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .

}
```

Listing 45: Index 6

```
SELECT * WHERE {
    ?product rdfs:label ?productLabel .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
    ?review dc:title ?revTitle .
}
```

Listing 46: Index 7

```
SELECT * WHERE {
    ?offer bsbm:product ?product .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .
```

```
    ?vendor rdfs:label ?vendorTitle .
    ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
    ?offer dc:publisher ?vendor .
    ?review bsbm:reviewFor ?product .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
}
```

Listing 47: Index 8

```
SELECT * WHERE {
    ?review bsbm:reviewFor ?product .
    ?review dc:title ?title .
    ?review rev:text ?text .
}
```

Listing 48: Index 9

```
SELECT * WHERE {
    ?review bsbm:reviewFor ?product .
    ?review bsbm:rating1 ?rating1 .
}
```

Listing 49: Index 10

2 Example Covers BSBM

Query1

```
Number of indices: 2
Patterns size: 4
Covered Triples:
?product rdfs:label ?label
?product rdf:type ?ProductType
?product bsbm:productFeature ?ProductFeature1
?product bsbm:productFeature ?ProductFeature2
Residual part: 1
Residual Triples:
?product bsbm:productPropertyNumeric1 ?value1
```

Listing 50: Cover 1

```
Number of indices: 2
Patterns size: 4
Covered Triples:
?product rdfs:label ?label
?product rdf:type ?ProductType
?product bsbm:productFeature ?ProductFeature1
?product bsbm:productFeature ?ProductFeature2
Residual part: 1
Residual Triples:
```

```
?product bsbm:productPropertyNumeric1 ?value1
```

Listing 51: Cover 2

```
Number of indices: 2
Patterns size: 4
Covered Triples:
?product rdfs:label ?label
?product rdf:type ?ProductType
?product bsbm:productFeature ?ProductFeature2
?product bsbm:productFeature ?ProductFeature1
Residual part: 1
Residual Triples:
?product bsbm:productPropertyNumeric1 ?value1
```

Listing 52: Cover 3

```
Number of indices: 1
Patterns size: 3
Covered Triples:
?product rdfs:label ?label
?product rdf:type ?ProductType
?product bsbm:productFeature ?ProductFeature1
Residual part: 2
Residual Triples:
?product bsbm:productFeature ?ProductFeature2
?product bsbm:productPropertyNumeric1 ?value1
```

Listing 53: Cover 4

```
Number of indices: 1
Patterns size: 3
Covered Triples:
?product rdfs:label ?label
?product rdf:type ?ProductType
?product bsbm:productFeature ?ProductFeature2
Residual part: 2
Residual Triples:
?product bsbm:productFeature ?ProductFeature1
?product bsbm:productPropertyNumeric1 ?value1
```

Listing 54: Cover 5

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?product rdf:type ?ProductType
?product bsbm:productFeature ?ProductFeature1
?product bsbm:productFeature ?ProductFeature2
Residual part: 2
Residual Triples:
?product rdfs:label ?label
?product bsbm:productPropertyNumeric1 ?value1
```

Listing 55: Cover 6

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?product rdf:type ?ProductType
?product bsbm:productFeature ?ProductFeature1
Residual part: 3
Residual Triples:
?product rdfs:label ?label
?product bsbm:productFeature ?ProductFeature2
?product bsbm:productPropertyNumeric1 ?value1
```

Listing 56: Cover 7

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?product rdf:type ?ProductType
?product bsbm:productFeature ?ProductFeature2
Residual part: 3
Residual Triples:
?product rdfs:label ?label
?product bsbm:productFeature ?ProductFeature1
?product bsbm:productPropertyNumeric1 ?value1
```

Listing 57: Cover 8

Query2

```
Number of indices: 2
Patterns size: 9
Covered Triples:
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
?product bsbm:productFeature ?f
?f rdfs:label ?productFeature
Residual part: 3
Residual Triples:
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 58: Cover 1

```
Number of indices: 3
Patterns size: 9
Covered Triples:
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
?product bsbm:productFeature ?f
?f rdfs:label ?productFeature
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
Residual part: 3
Residual Triples:
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 59: Cover 2

```
Number of indices: 3
Patterns size: 9
Covered Triples:
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
?product bsbm:productFeature ?f
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
?f rdfs:label ?productFeature
Residual part: 3
Residual Triples:
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 60: Cover 3

```
Number of indices: 2
Patterns size: 8
Covered Triples:
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
?product bsbm:productFeature ?f
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
Residual part: 4
Residual Triples:
```

```
?f rdfs:label ?productFeature
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 61: Cover 4

```
Number of indices: 1
Patterns size: 7
Covered Triples:
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
Residual part: 5
Residual Triples:
?product bsbm:productFeature ?f
?f rdfs:label ?productFeature
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 62: Cover 5

```
Number of indices: 2
Patterns size: 5
Covered Triples:
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
?product bsbm:productFeature ?f
?f rdfs:label ?productFeature
Residual part: 7
Residual Triples:
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 63: Cover 6

```
Number of indices: 2
Patterns size: 5
Covered Triples:
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
```

```
?product bsbm:productFeature ?f
?f rdfs:label ?productFeature
Residual part: 7
Residual Triples:
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 64: Cover 7

```
Number of indices: 1
Patterns size: 4
Covered Triples:
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
?product bsbm:productFeature ?f
Residual part: 8
Residual Triples:
?product rdfs:label ?label
?product rdfs:comment ?comment
?f rdfs:label ?productFeature
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 65: Cover 8

```
Number of indices: 1
Patterns size: 3
Covered Triples:
?product dc:publisher ?p
?product bsbm:productFeature ?f
?f rdfs:label ?productFeature
Residual part: 9
Residual Triples:
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:producer ?p
?p rdfs:label ?producer
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 66: Cover 9

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?product bsbm:productFeature ?f
?f rdfs:label ?productFeature
Residual part: 10
Residual Triples:
?product rdfs:label ?label
?product rdfs:comment ?comment
?product bsbm:producer ?p
?p rdfs:label ?producer
?product dc:publisher ?p
?product bsbm:productPropertyTextual1 ?propertyTextual1
?product bsbm:productPropertyTextual2 ?propertyTextual2
?product bsbm:productPropertyTextual3 ?propertyTextual3
?product bsbm:productPropertyNumeric1 ?propertyNumeric1
?product bsbm:productPropertyNumeric2 ?propertyNumeric2
```

Listing 67: Cover 10

Query3

```
Number of indices: 3
Patterns size: 12
Covered Triples:
?product rdfs:label ?productLabel
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?review bsbm:rating1 ?rating1
Residual part: 1
Residual Triples:
?offer bsbm:validTo ?date
```

Listing 68: Cover 1

```
Number of indices: 4
Patterns size: 12
Covered Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?review bsbm:reviewFor ?product
```

```
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?offer bsbm:validTo ?date
Residual part: 1
Residual Triples:
?review bsbm:rating1 ?rating1
```

Listing 69: Cover 2

```
Number of indices: 2
Patterns size: 11
Covered Triples:
?product rdfs:label ?productLabel
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
Residual part: 2
Residual Triples:
?offer bsbm:validTo ?date
?review bsbm:rating1 ?rating1
```

Listing 70: Cover 3

```
Number of indices: 3
Patterns size: 11
Covered Triples:
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review bsbm:rating1 ?rating1
?offer bsbm:validTo ?date
Residual part: 2
Residual Triples:
?product rdfs:label ?productLabel
?review dc:title ?revTitle
```

Listing 71: Cover 4

```
Number of indices: 3
Patterns size: 11
Covered Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
Residual part: 2
Residual Triples:
?offer bsbm:validTo ?date
?review bsbm:rating1 ?rating1
```

Listing 72: Cover 5

```
Number of indices: 3
Patterns size: 11
Covered Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?offer bsbm:validTo ?date
Residual part: 2
Residual Triples:
?review dc:title ?revTitle
?review bsbm:rating1 ?rating1
```

Listing 73: Cover 6

```
Number of indices: 2
Patterns size: 10
Covered Triples:
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
```

```
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review bsbm:rating1 ?rating1
Residual part: 3
Residual Triples:
?product rdfs:label ?productLabel
?offer bsbm:validTo ?date
?review dc:title ?revTitle
```

Listing 74: Cover 7

```
Number of indices: 2
Patterns size: 10
Covered Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
Residual part: 3
Residual Triples:
?offer bsbm:validTo ?date
?review dc:title ?revTitle
?review bsbm:rating1 ?rating1
```

Listing 75: Cover 8

```
Number of indices: 2
Patterns size: 10
Covered Triples:
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?offer bsbm:validTo ?date
Residual part: 3
Residual Triples:
?product rdfs:label ?productLabel
?review dc:title ?revTitle
?review bsbm:rating1 ?rating1
```

Listing 76: Cover 9

```
Number of indices: 1
Patterns size: 9
Covered Triples:
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
Residual part: 4
Residual Triples:
?product rdfs:label ?productLabel
?offer bsbm:validTo ?date
?review dc:title ?revTitle
?review bsbm:rating1 ?rating1
```

Listing 77: Cover 10

```
Number of indices: 2
Patterns size: 8
Covered Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
Residual part: 5
Residual Triples:
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?offer bsbm:validTo ?date
?review bsbm:rating1 ?rating1
```

Listing 78: Cover 11

```
Number of indices: 2
Patterns size: 6
Covered Triples:
?product rdfs:label ?productLabel
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
?review bsbm:rating1 ?rating1
Residual part: 7
Residual Triples:
```

```
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?offer bsbm:validTo ?date
```

Listing 79: Cover 12

```
Number of indices: 1
Patterns size: 5
Covered Triples:
?product rdfs:label ?productLabel
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
Residual part: 8
Residual Triples:
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?offer bsbm:validTo ?date
?review bsbm:rating1 ?rating1
```

Listing 80: Cover 13

```
Number of indices: 2
Patterns size: 5
Covered Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?offer bsbm:validTo ?date
Residual part: 8
Residual Triples:
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
?review bsbm:rating1 ?rating1
```

Listing 81: Cover 14

```
Number of indices: 1
```

```
Patterns size: 4
Covered Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
Residual part: 9
Residual Triples:
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?offer bsbm:validTo ?date
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
?review bsbm:rating1 ?rating1
```

Listing 82: Cover 15

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?review bsbm:reviewFor ?product
?review bsbm:rating1 ?rating1
Residual part: 11
Residual Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?offer bsbm:vendor ?vendor
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
?offer dc:publisher ?vendor
?offer bsbm:validTo ?date
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
```

Listing 83: Cover 16

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?offer bsbm:vendor ?vendor
?offer bsbm:validTo ?date
Residual part: 11
Residual Triples:
?product rdfs:label ?productLabel
?offer bsbm:product ?product
?offer bsbm:price ?price
?vendor rdfs:label ?vendorTitle
?vendor bsbm:country http://downlode.org/rdf/iso-3166/countries#DE
```

```
?offer dc:publisher ?vendor
?review bsbm:reviewFor ?product
?review rev:reviewer ?reviewer
?reviewer foaf:name ?revName
?review dc:title ?revTitle
?review bsbm:rating1 ?rating1
```

Listing 84: Cover 17

Appendix B Queries and Indexes

3 SPARQL Performance Benchmark

All queries and indices of SPARQL Performance Benchmark use the following namespaces:

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrc:<http://swrc.ontoware.org/ontology#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX bench:<http://localhost/vocabulary/bench/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX dcterms:<http://purl.org/dc/terms/>
```

Listing 85: Common namespace prefixes

3.1 Queries

```
SELECT * WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
}
```

Listing 86: Query 1

```
SELECT * WHERE {
  ?article rdf:type bench:Article .
  ?article dc:title ?title .
  ?article swrc:journal ?journal .
  ?article swrc:month   ?month.
  ?article swrc:pages   ?pages.
}
```

Listing 87: Query 2

```
SELECT * WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
```

```
  ?document dc:title ?title.
}
```

Listing 88: Query 3

```
SELECT * WHERE {
  ?journal rdf:type bench:Journal .
  ?journal dc:title ?title .
  ?journal dcterms:issued ?yr
}
```

Listing 89: Query 4

```
SELECT * WHERE {
  ?article rdf:type bench:Article .
  ?article swrc:pages ?value
}
```

Listing 90: Query 5

```
SELECT * WHERE {
  ?article rdf:type bench:Article .
  ?article swrc:month ?value
}
```

Listing 91: Query 6

```
SELECT * WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person2 .
  ?person foaf:name ?name .
  ?person2 foaf:name ?name
}
```

Listing 92: Query 7

```
SELECT * WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person .
  ?person foaf:name ?name
}
```

Listing 93: Query 8

```
SELECT * WHERE {
  ?erdoes rdf:type foaf:Person .
  ?erdoes foaf:name ?name .
  ?document dc:creator ?erdoes .
```

```
  ?document dc:creator ?author .
  ?document2 dc:creator ?author .
  ?document2 dc:creator ?author2 .
  ?author2 foaf:name ?name
}
```

Listing 94: Query 9

```
SELECT * WHERE {
  ?journal rdf:type bench:Journal .
  ?journal dc:title ?title .
}
```

Listing 95: Query 10

```
SELECT * WHERE {
  ?journal rdf:type bench:Journal .
  ?journal dcterms:issued ?yr
}
```

Listing 96: Query 11

```
SELECT * WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
}
```

Listing 97: Query 12

```
SELECT * WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
}
```

Listing 98: Query 13

```
SELECT * WHERE {
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
}
```

Listing 99: Query 14

```
SELECT * WHERE {
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
```

```
  ?inproc foaf:homepage ?url .
}
```

Listing 100: Query 15

```
SELECT * WHERE {
  ?erdoes rdf:type foaf:Person .
  ?erdoes foaf:name ?name .
  ?document dc:creator ?erdoes .
  ?document dc:creator ?author .
  ?document2 dc:creator ?author .
  ?document2 dc:creator ?author2 .
}
```

Listing 101: Query 16

3.2 Indexes

```
SELECT * WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
}
```

Listing 102: Index 1

```
SELECT * WHERE {
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
}
```

Listing 103: Index 2

```
SELECT ?inproc ?author WHERE {
   ?inproc dc:title ?title .
  ?inproc swrc:pages ?page .
}
```

Listing 104: Index 3

```
SELECT ?inproc ?author WHERE {
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
}
```

Listing 105: Index 4

```
SELECT * WHERE {
  ?article rdf:type bench:Article .
  ?article dc:title ?title .
 }
```

Listing 106: Index 5

```
SELECT * WHERE {
  ?article dc:title ?title .
  ?article swrc:journal ?journal .
}
```

Listing 107: Index 6

```
SELECT * WHERE {
  ?article dc:title ?title .
  ?article swrc:month   ?month.
}
```

Listing 108: Index 7

```
SELECT * WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
}
```

Listing 109: Index 8

```
SELECT * WHERE {
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
}
```

Listing 110: Index 9

```
SELECT * WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
}
```

Listing 111: Index 10

4 Example Covers SPARQL Performance Benchmark

Query1

```
Number of indices: 2
Patterns size: 6
Covered Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @dc:title ?title
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://swrc.ontoware.org/ontology#pages ?page
Residual part: 2
Residual Triples:
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 112: Cover 1

```
Number of indices: 3
Patterns size: 6
Covered Triples:
?inproc @dc:title ?title
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
Residual part: 2
Residual Triples:
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 113: Cover 2

```
Number of indices: 2
Patterns size: 5
Covered Triples:
?inproc @dc:title ?title
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
Residual part: 3
Residual Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 114: Cover 3

```
Number of indices: 2
Patterns size: 5
Covered Triples:
?inproc @dc:title ?title
```

```
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
Residual part: 3
Residual Triples:
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 115: Cover 4

```
Number of indices: 3
Patterns size: 5
Covered Triples:
?inproc @dc:title ?title
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @rdfs:seeAlso ?ee
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
Residual part: 3
Residual Triples:
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 116: Cover 5

```
Number of indices: 1
Patterns size: 4
Covered Triples:
?inproc @dc:title ?title
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://swrc.ontoware.org/ontology#pages ?page
Residual part: 4
Residual Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 117: Cover 6

```
Number of indices: 2
Patterns size: 4
Covered Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @dc:title ?title
?inproc @http://swrc.ontoware.org/ontology#pages ?page
Residual part: 4
Residual Triples:
```

```
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 118: Cover 7

```
Number of indices: 2
Patterns size: 4
Covered Triples:
?inproc @dc:title ?title
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @rdfs:seeAlso ?ee
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
Residual part: 4
Residual Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 119: Cover 8

```
Number of indices: 1
Patterns size: 3
Covered Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @dc:title ?title
Residual part: 5
Residual Triples:
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 120: Cover 9

```
Number of indices: 1
Patterns size: 3
Covered Triples:
?inproc @rdfs:seeAlso ?ee
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
Residual part: 5
Residual Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @dc:title ?title
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 121: Cover 10

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?inproc @dc:title ?title
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
Residual part: 5
Residual Triples:
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 122: Cover 11

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?inproc @dc:title ?title
?inproc @http://swrc.ontoware.org/ontology#pages ?page
Residual part: 6
Residual Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 123: Cover 12

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?inproc @rdf:type http://localhost/vocabulary/bench/Inproceedings
?inproc @dc:title ?title
Residual part: 6
Residual Triples:
?inproc @http://localhost/vocabulary/bench/booktitle ?booktitle
?inproc @http://purl.org/dc/terms/partOf ?proc
?inproc @rdfs:seeAlso ?ee
?inproc @http://swrc.ontoware.org/ontology#pages ?page
?inproc @http://xmlns.com/foaf/0.1/homepage ?url
?inproc @http://purl.org/dc/terms/issued ?yr
```

Listing 124: Cover 13

Query2

```
Number of indices: 4
Patterns size: 5
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
Residual part: 0
Residual Triples:
```

Listing 125: Cover 1

```
Number of indices: 4
Patterns size: 5
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
?article @rdf:type http://localhost/vocabulary/bench/Article
Residual part: 0
Residual Triples:
```

Listing 126: Cover 2

```
Number of indices: 3
Patterns size: 4
Covered Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
Residual part: 1
Residual Triples:
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 127: Cover 3

```
Number of indices: 3
Patterns size: 4
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
Residual part: 1
Residual Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
```

Listing 128: Cover 4

```
Number of indices: 3
Patterns size: 4
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
?article @rdf:type http://localhost/vocabulary/bench/Article
Residual part: 1
Residual Triples:
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 129: Cover 5

```
Number of indices: 3
Patterns size: 4
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @http://swrc.ontoware.org/ontology#month ?month
?article @rdf:type http://localhost/vocabulary/bench/Article
Residual part: 1
Residual Triples:
?article @http://swrc.ontoware.org/ontology#journal ?journal
```

Listing 130: Cover 6

```
Number of indices: 3
Patterns size: 4
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @http://swrc.ontoware.org/ontology#journal ?journal
Residual part: 1
Residual Triples:
?article @http://swrc.ontoware.org/ontology#month ?month
```

Listing 131: Cover 7

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#month ?month
Residual part: 2
Residual Triples:
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 132: Cover 8

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
Residual part: 2
Residual Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 133: Cover 9

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @http://swrc.ontoware.org/ontology#month ?month
Residual part: 2
Residual Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @http://swrc.ontoware.org/ontology#journal ?journal
```

Listing 134: Cover 10

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#month ?month
?article @rdf:type http://localhost/vocabulary/bench/Article
Residual part: 2
Residual Triples:
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 135: Cover 11

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#journal ?journal
Residual part: 2
Residual Triples:
?article @http://swrc.ontoware.org/ontology#month ?month
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 136: Cover 12

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @rdf:type http://localhost/vocabulary/bench/Article
Residual part: 2
Residual Triples:
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
```

Listing 137: Cover 13

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @http://swrc.ontoware.org/ontology#journal ?journal
Residual part: 2
Residual Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @http://swrc.ontoware.org/ontology#month ?month
```

Listing 138: Cover 14

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @rdf:type http://localhost/vocabulary/bench/Article
Residual part: 2
Residual Triples:
?article @http://swrc.ontoware.org/ontology#month ?month
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 139: Cover 15

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
?article @rdf:type http://localhost/vocabulary/bench/Article
Residual part: 2
Residual Triples:
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
```

Listing 140: Cover 16

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#month ?month
Residual part: 3
Residual Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 141: Cover 17

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @dc:title ?title
Residual part: 3
Residual Triples:
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 142: Cover 18

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#journal ?journal
Residual part: 3
Residual Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @http://swrc.ontoware.org/ontology#month ?month
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 143: Cover 19

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?article @dc:title ?title
?article @http://swrc.ontoware.org/ontology#pages ?pages
Residual part: 3
Residual Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
```

Listing 144: Cover 20

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?article @rdf:type http://localhost/vocabulary/bench/Article
?article @dc:title ?title
Residual part: 3
Residual Triples:
?article @http://swrc.ontoware.org/ontology#journal ?journal
?article @http://swrc.ontoware.org/ontology#month ?month
?article @http://swrc.ontoware.org/ontology#pages ?pages
```

Listing 145: Cover 21

Query3

```
Number of indices: 2
Patterns size: 4
Covered Triples:
?class @rdfs:subClassOf http://xmlns.com/foaf/0.1/Document
?document @rdf:type ?class
?document @http://purl.org/dc/terms/issued ?yr
?document @dc:title ?title
Residual part: 0
Residual Triples:
```

Listing 146: Cover 1

```
Number of indices: 1
Patterns size: 3
Covered Triples:
?class @rdfs:subClassOf http://xmlns.com/foaf/0.1/Document
?document @rdf:type ?class
?document @http://purl.org/dc/terms/issued ?yr
Residual part: 1
Residual Triples:
?document @dc:title ?title
```

Listing 147: Cover 2

```
Number of indices: 2
Patterns size: 3
Covered Triples:
?class @rdfs:subClassOf http://xmlns.com/foaf/0.1/Document
?document @rdf:type ?class
?document @dc:title ?title
Residual part: 1
Residual Triples:
?document @http://purl.org/dc/terms/issued ?yr
```

Listing 148: Cover 3

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?class @rdfs:subClassOf http://xmlns.com/foaf/0.1/Document
?document @rdf:type ?class
Residual part: 2
Residual Triples:
?document @http://purl.org/dc/terms/issued ?yr
?document @dc:title ?title
```

Listing 149: Cover 4

```
Number of indices: 1
Patterns size: 2
Covered Triples:
?document @rdf:type ?class
?document @dc:title ?title
Residual part: 2
Residual Triples:
?class @rdfs:subClassOf http://xmlns.com/foaf/0.1/Document
?document @http://purl.org/dc/terms/issued ?yr
```

Listing 150: Cover 5

Appendix C Additional Evaluation

5 RDFMatView Evaluation: BSBM and SP2B Benchmarks

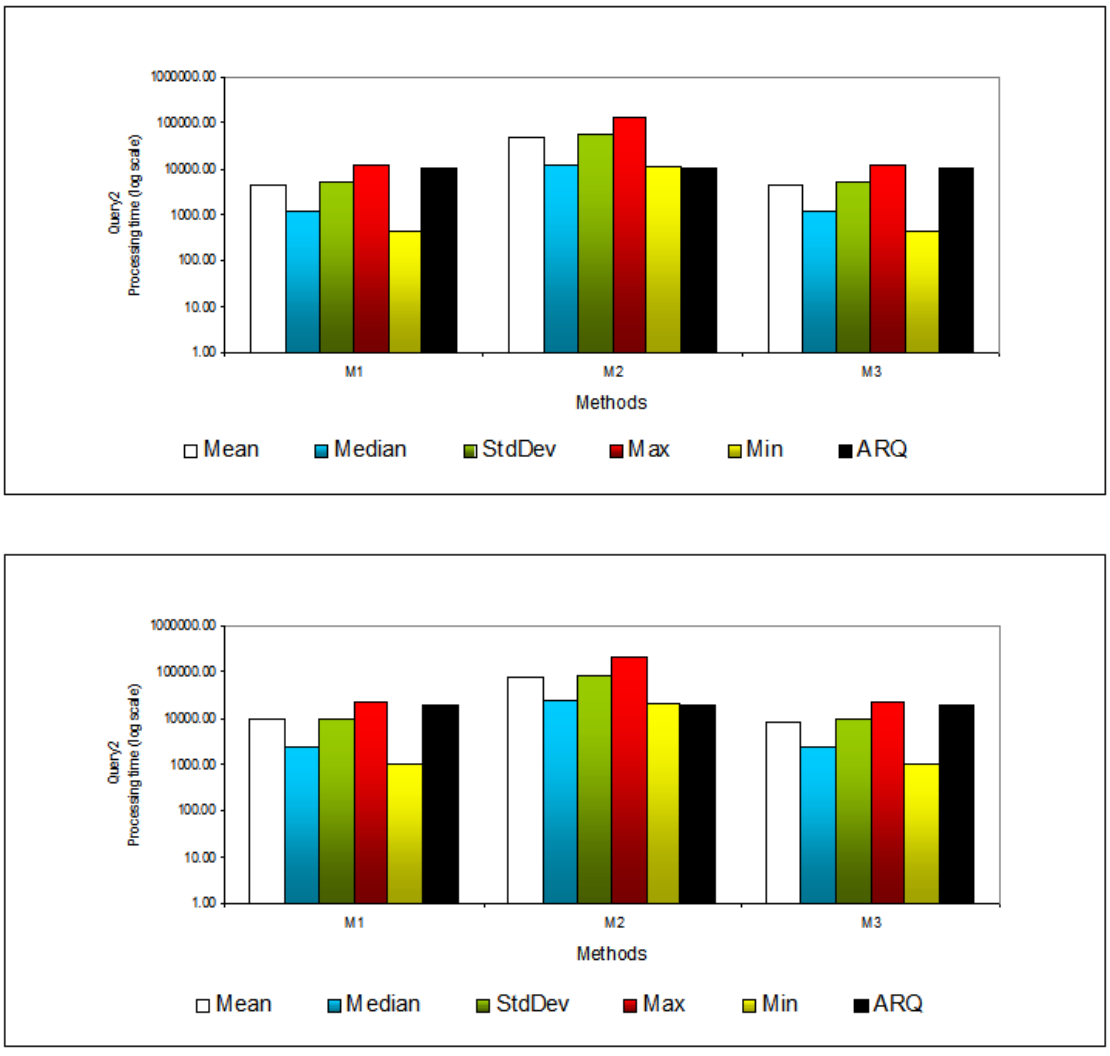


Figure 1: Processing of query2 over BSBM using 250K and 500K triples datasets. Results show that rewriting methods M1 and M3 are more efficient than M2.

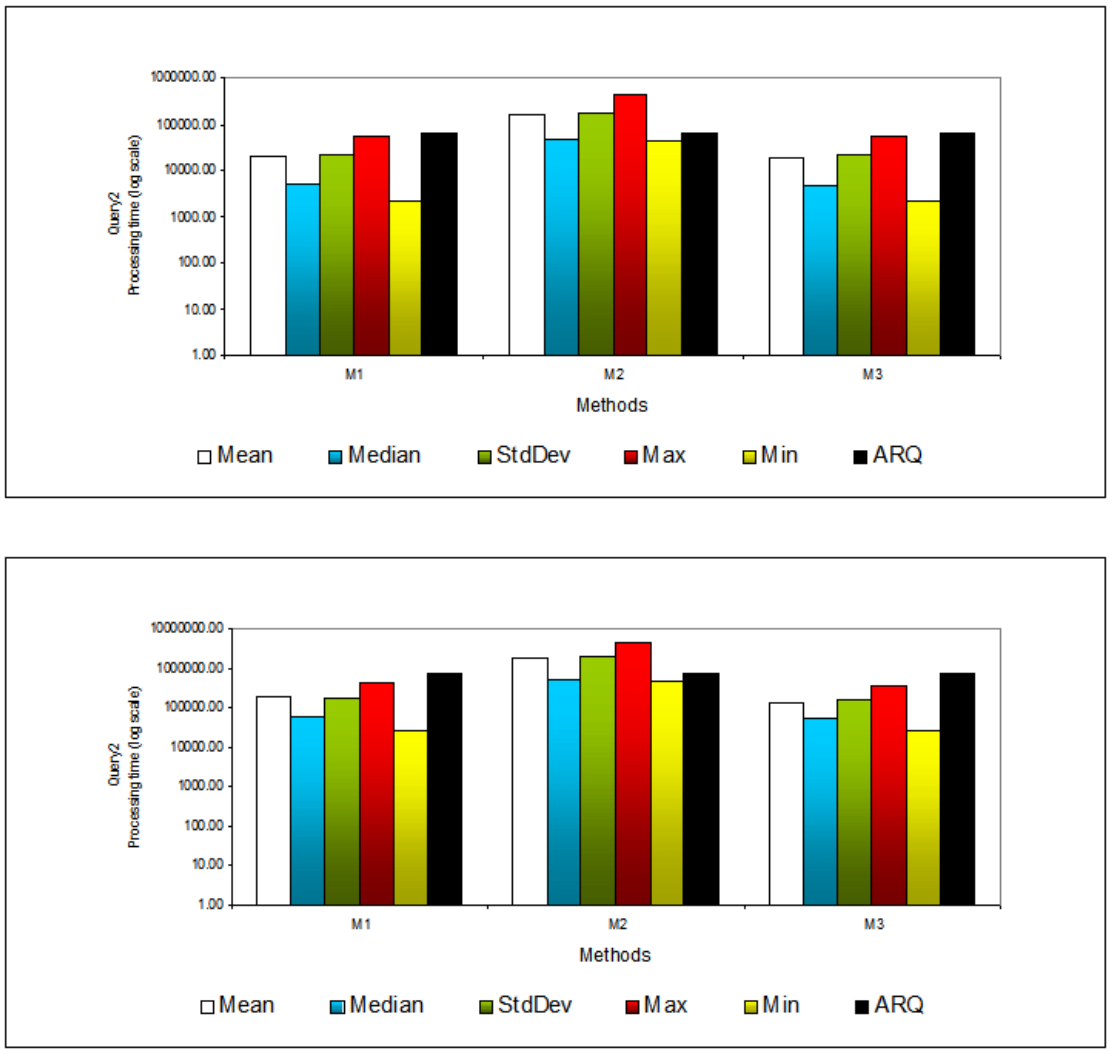


Figure 2: Processing of query2 over BSBM using 1M and 10M triples datasets. For explanation see Figure 1.

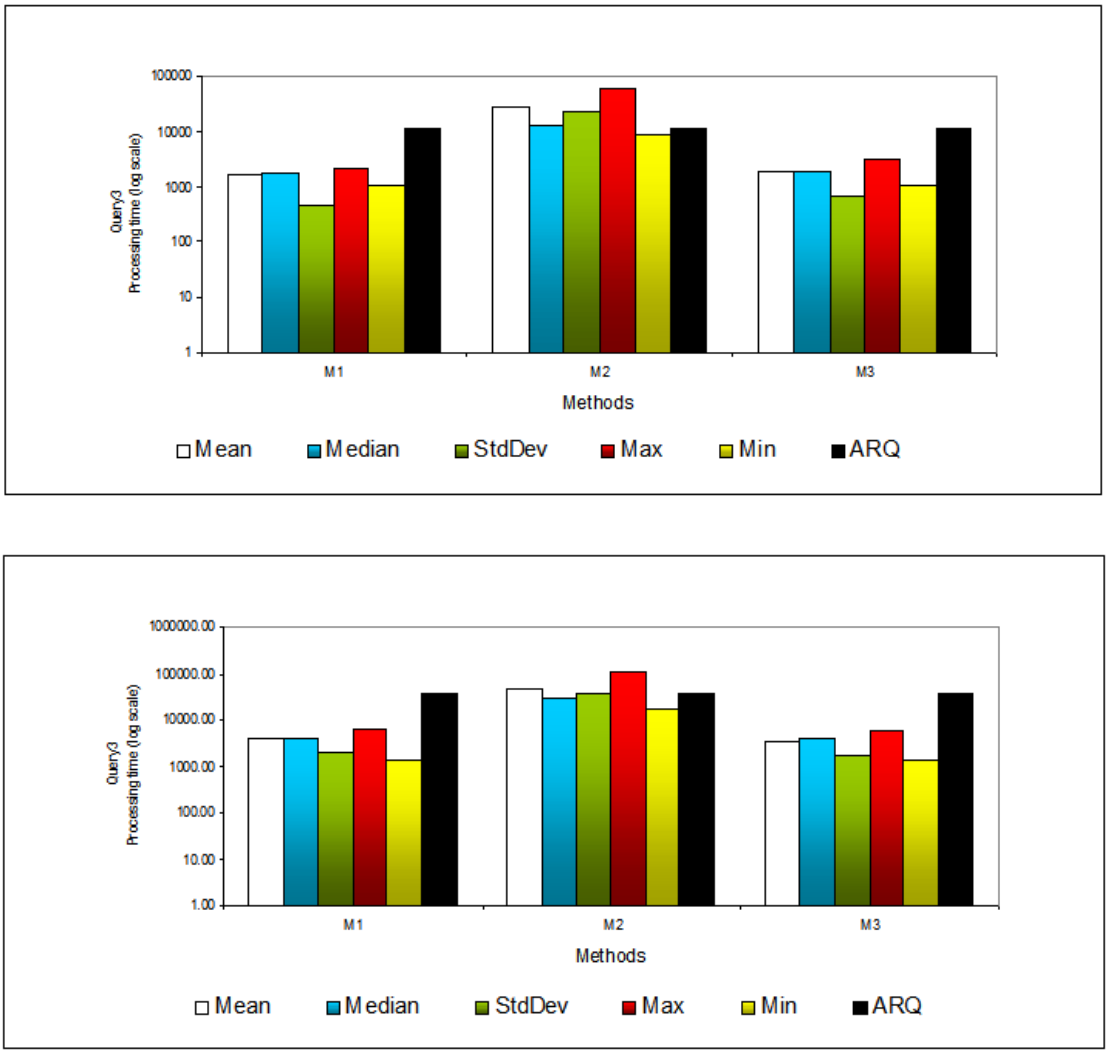


Figure 3: Processing of query3 over BSBM using 250K and 500K triples datasets. For explanation see Figure 1.

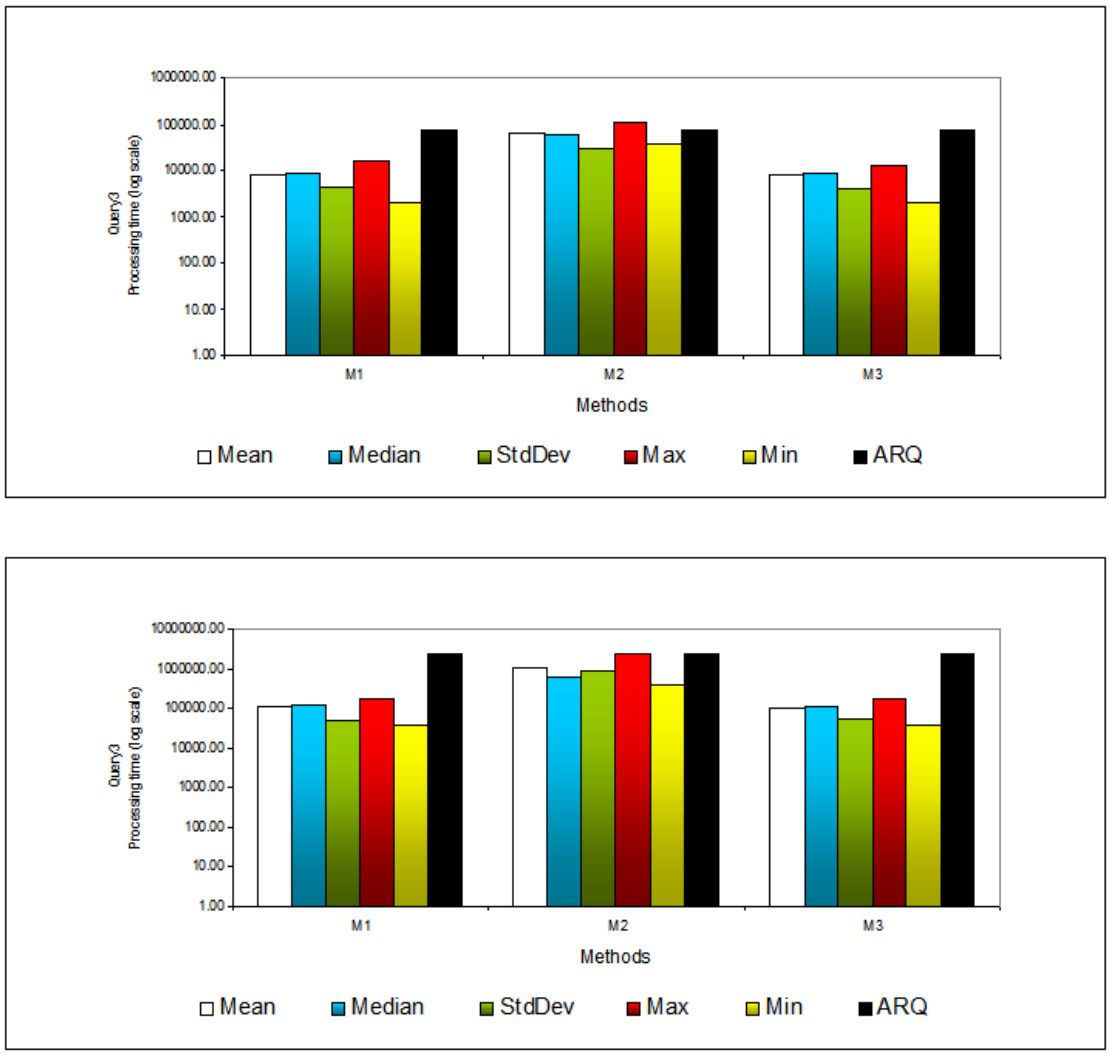


Figure 4: Processing of query3 over BSBM using 1M and 10M triples datasets. For explanation see Figure 1.

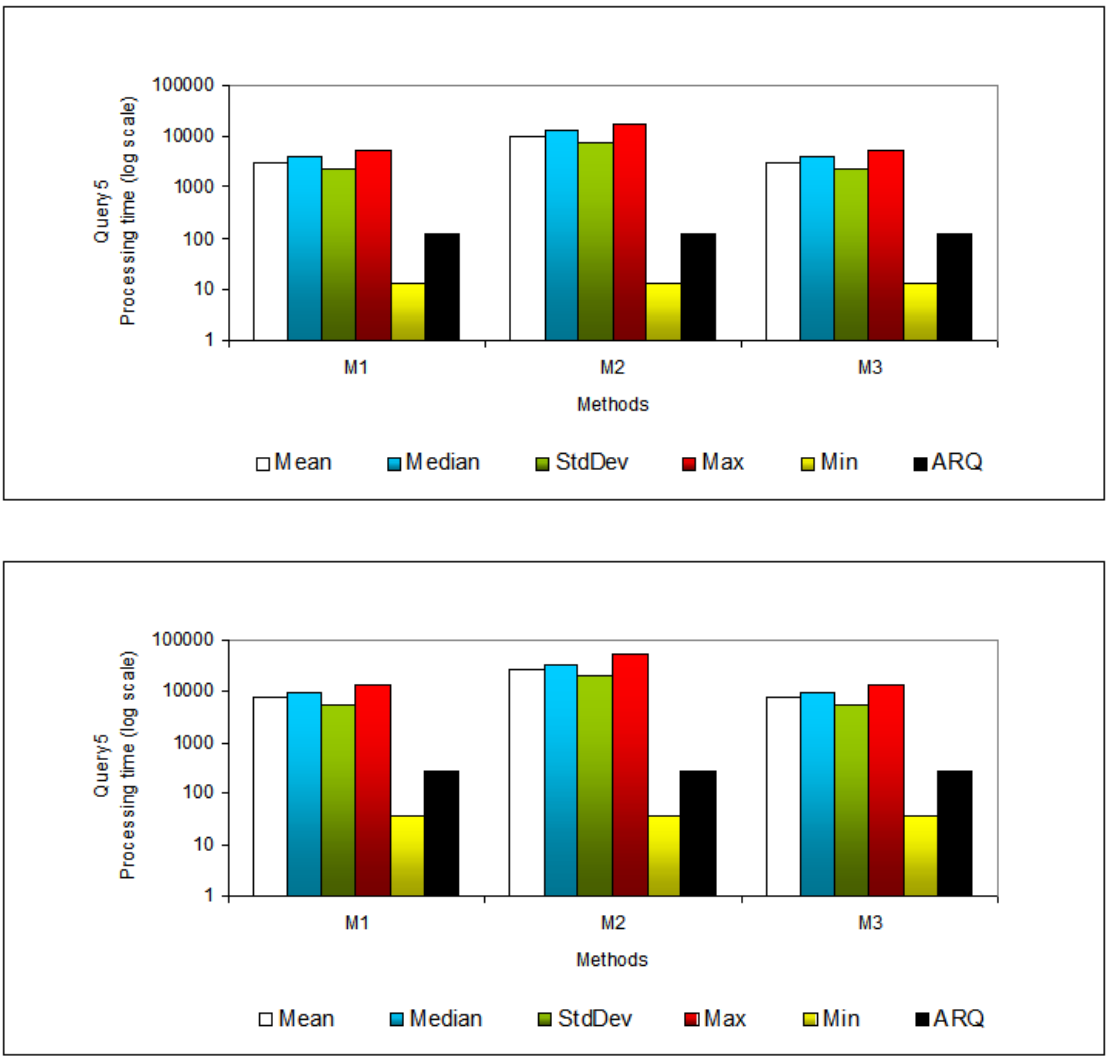


Figure 5: Processing of query5 over SP2B using 250K and 500K triples datasets. Results show that all rewriting methods are capable to improve standard processing time, depending on the cover selected to execute the query. However, similar to Figure 2.11 processing time using method M2 is, in general, significantly larger than that of M1 and M3.

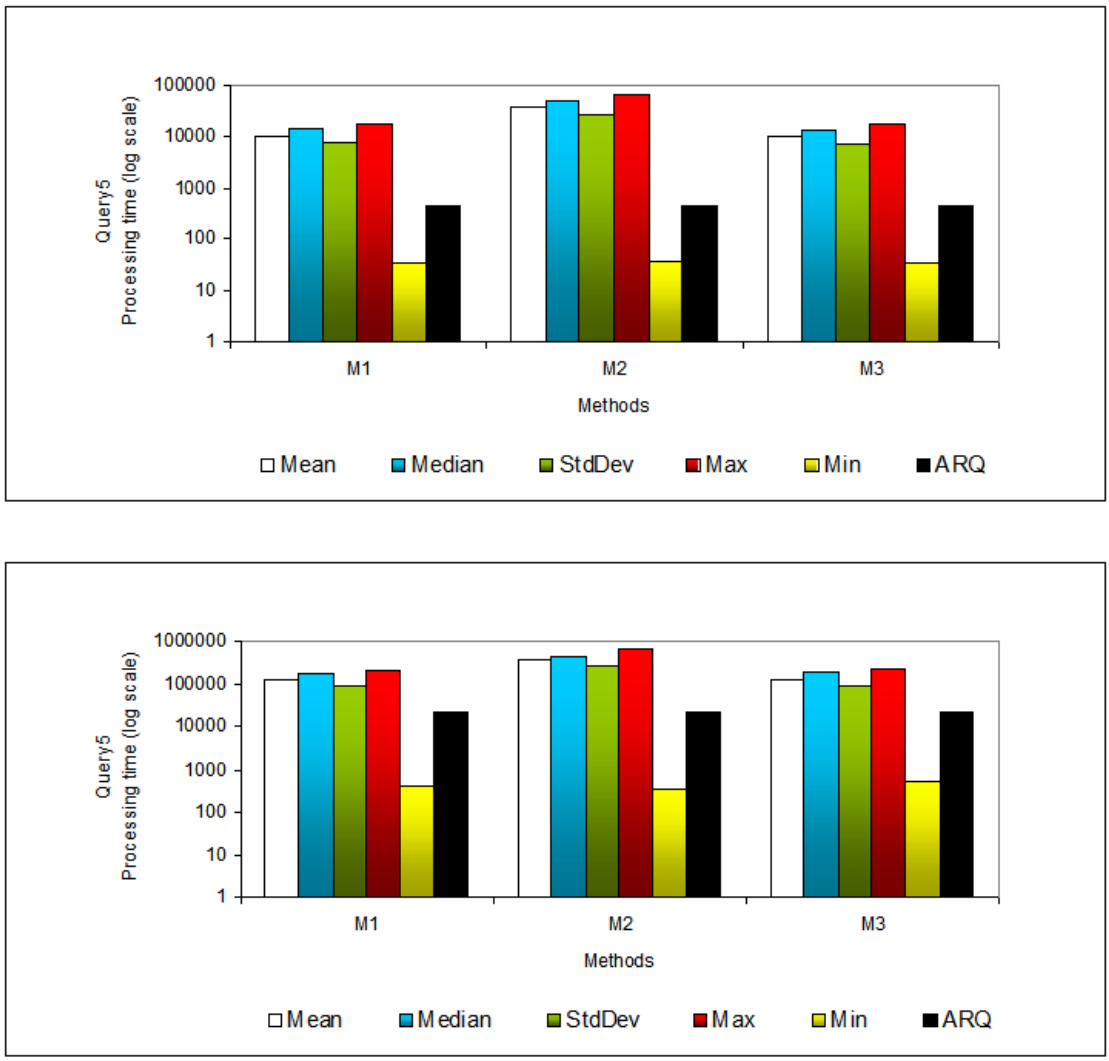


Figure 6: Processing of query5 over SP2B using 1M and 10M triples datasets. For explanation see Figure 5.

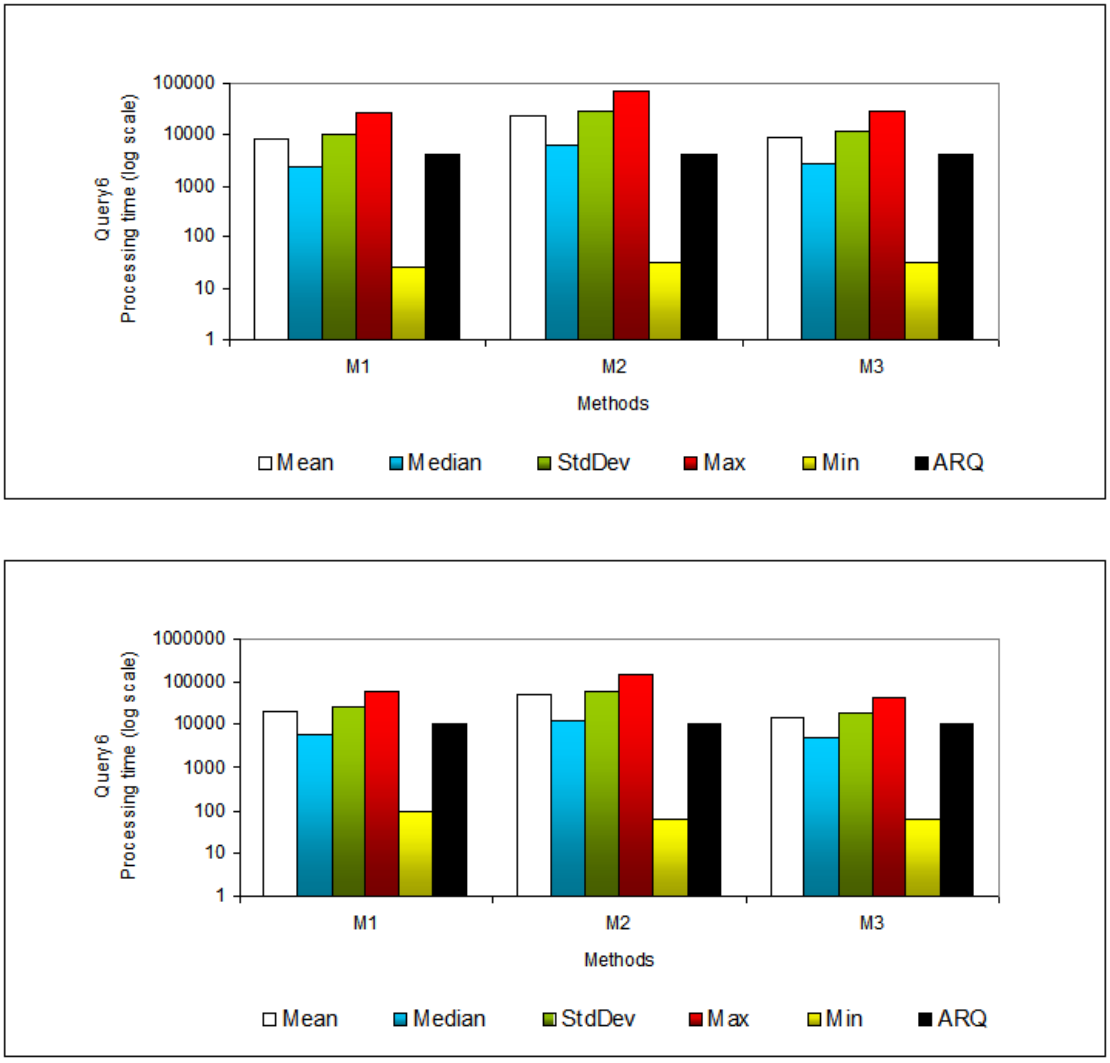


Figure 7: Processing of query6 over SP2B using 250K and 500K triples datasets. For explanation see Figure 5.

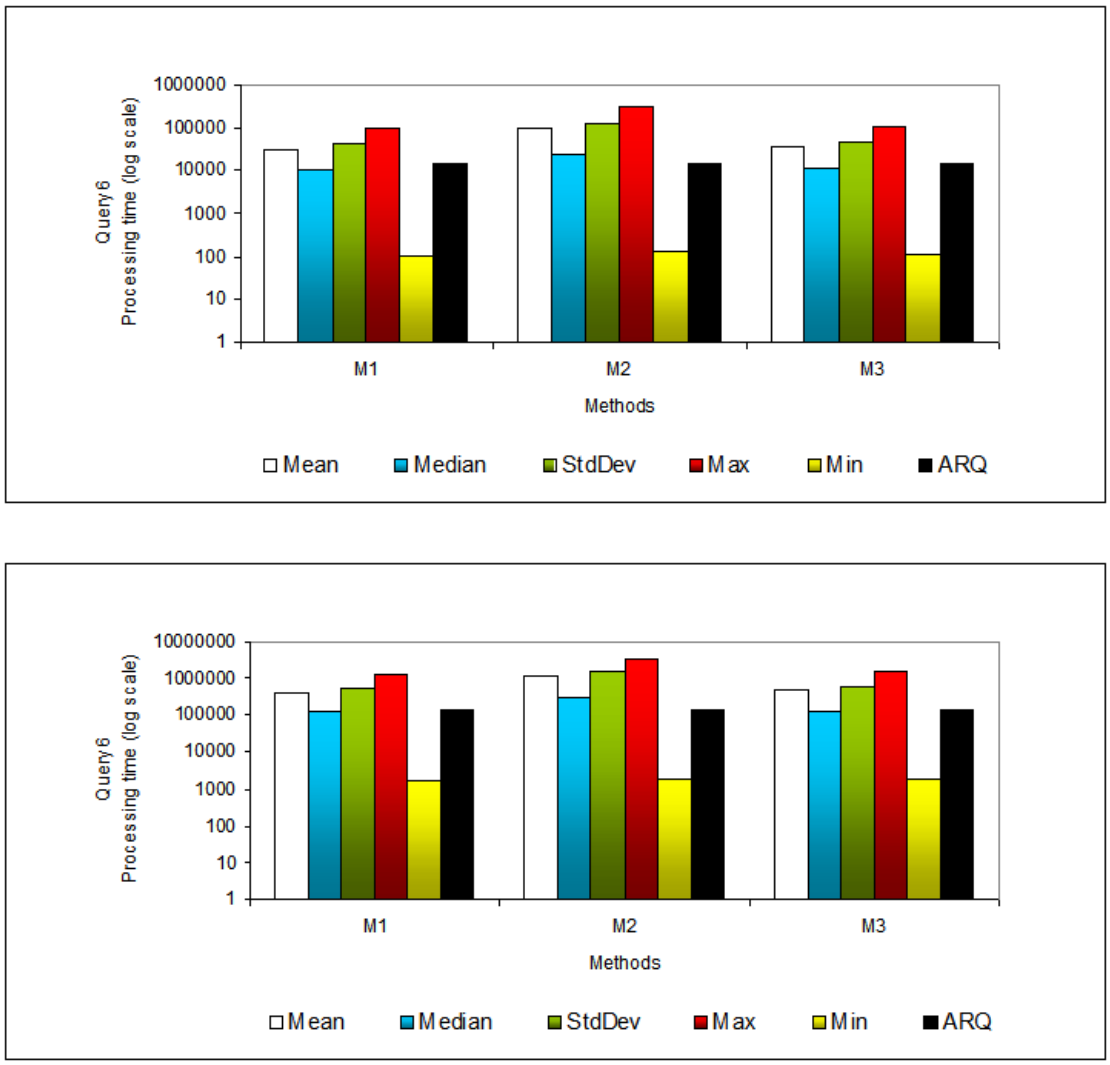
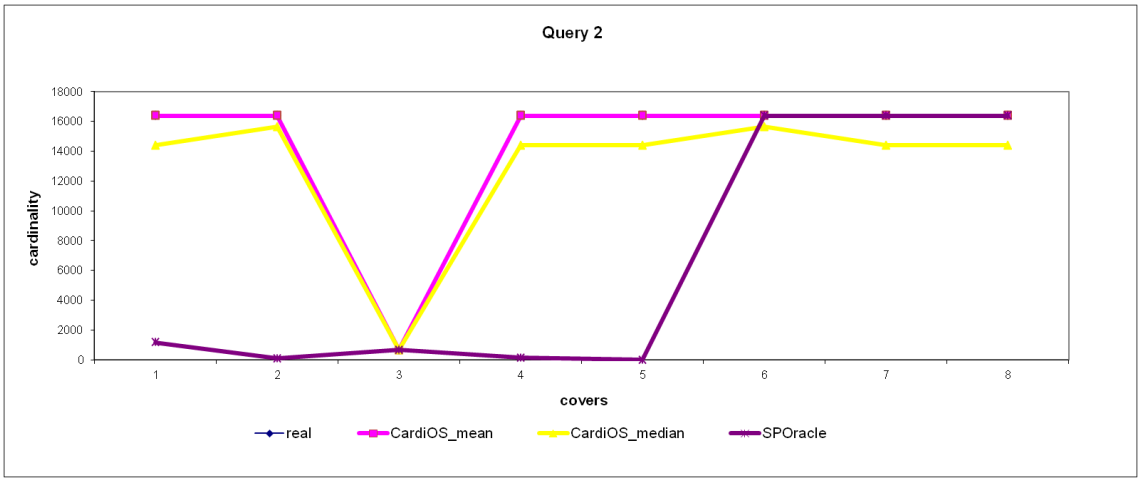
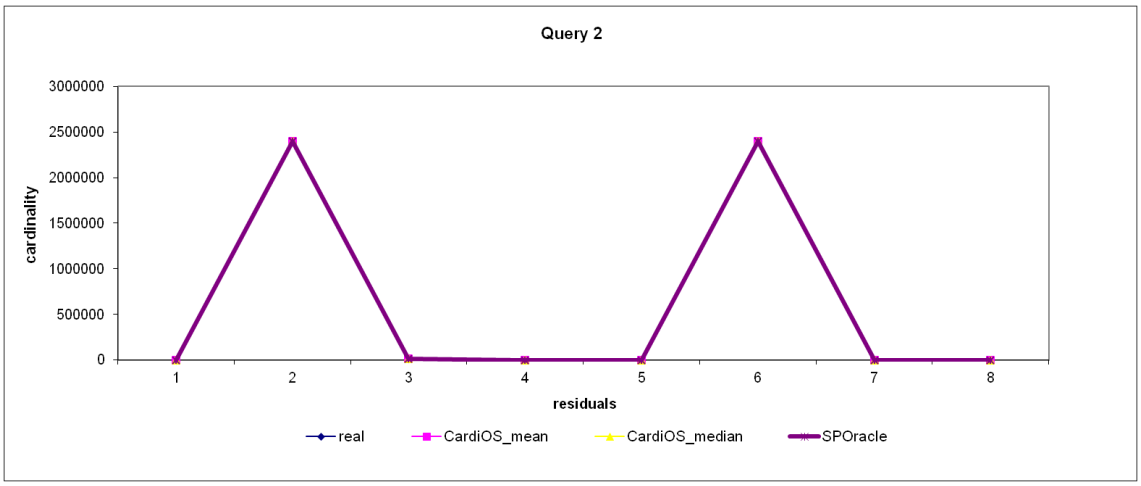


Figure 8: Processing of query6 over SP2B using 1M and 10M triples datasets. For explanation see Figure 5.

6 Cardinality Estimation: Covered and Residual Patterns

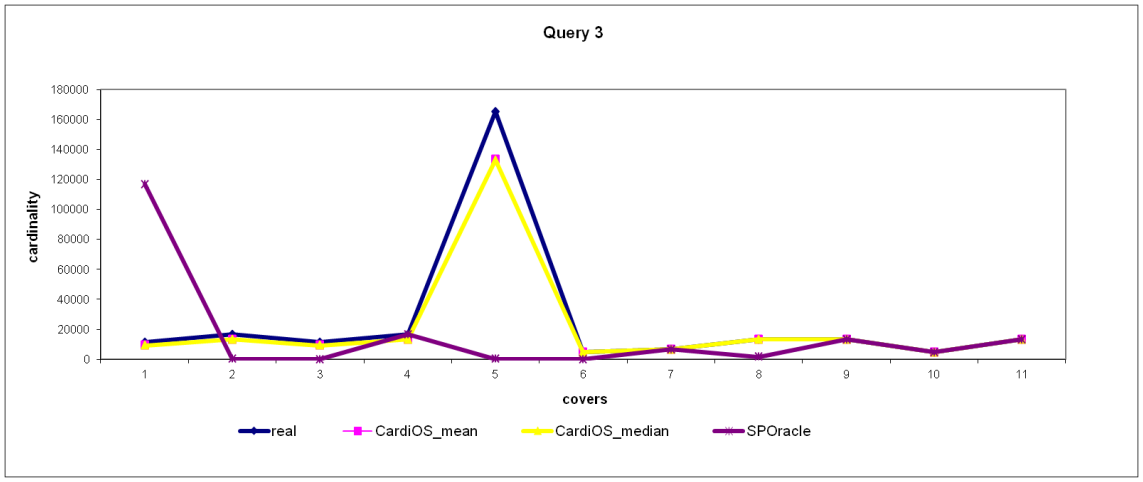


(a) Covers of query 2.

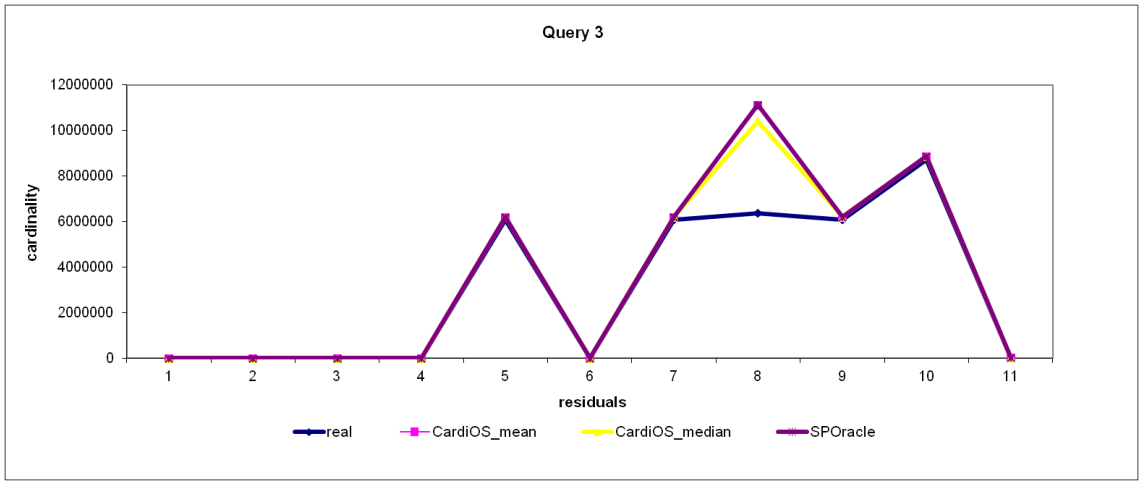


(b) Residuals of query 2.

Figure 9: Evaluation of cardinality estimation on BSBM queries.

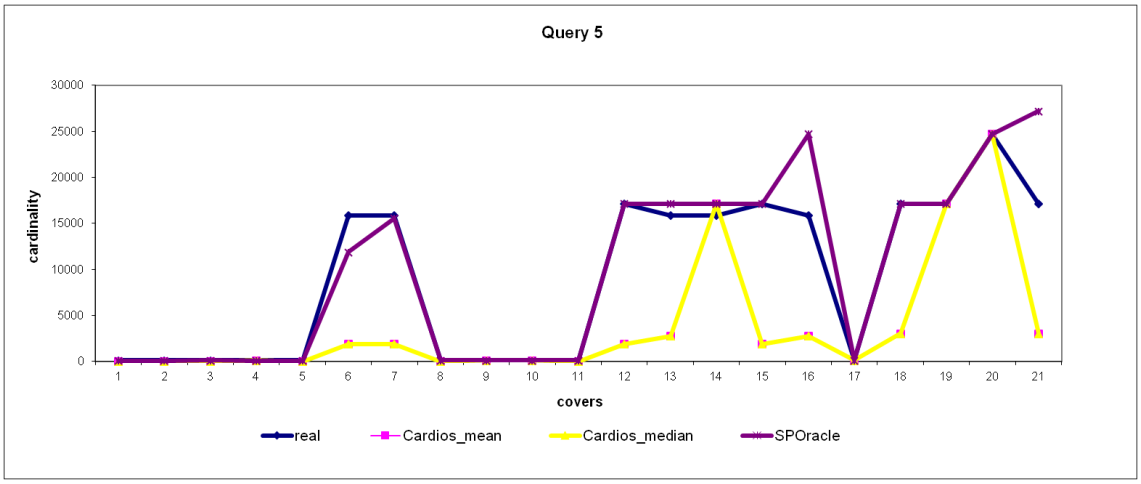


(a) Covers of query 3.

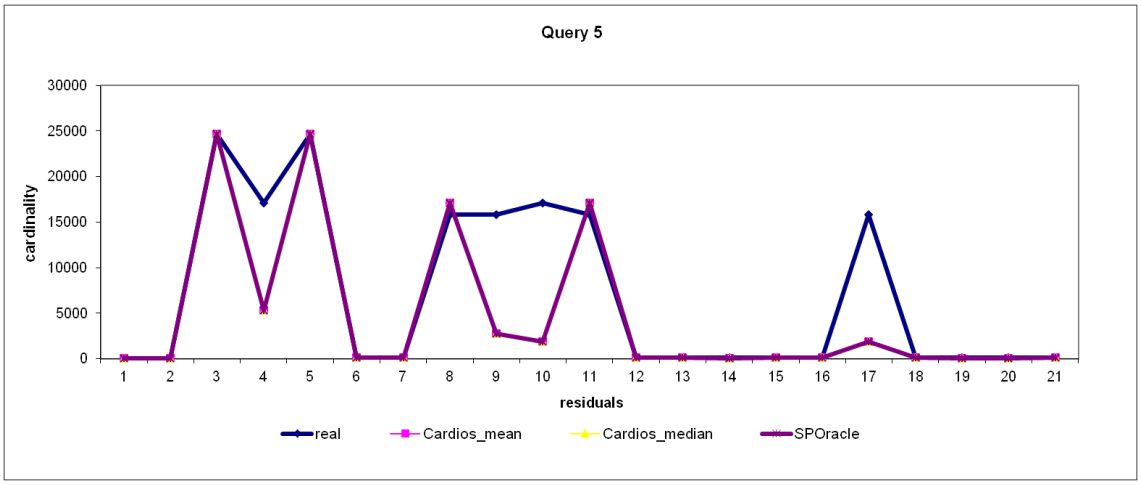


(b) Residuals of query 3.

Figure 10: Evaluation of cardinality estimation on BSBM queries.

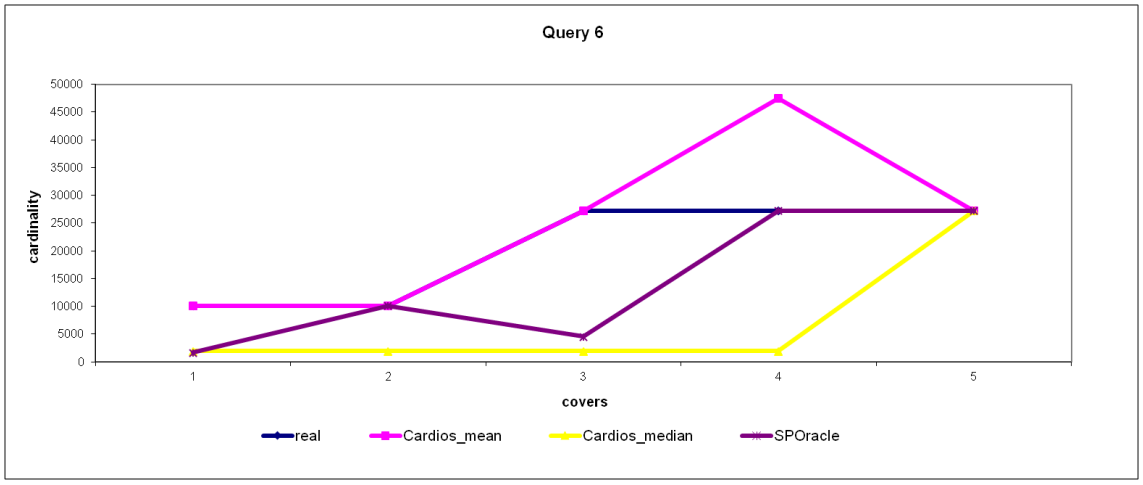


(a) Covers of query 5.

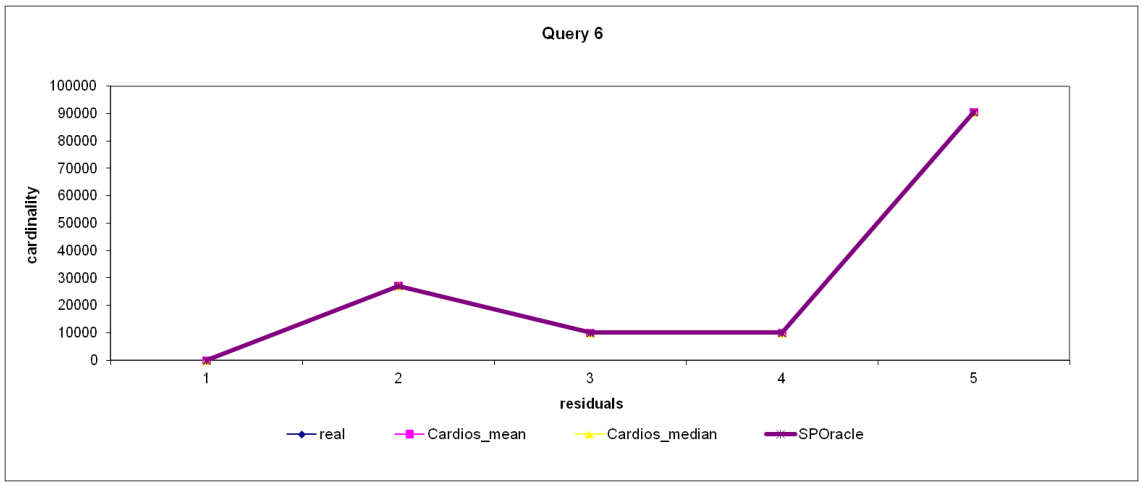


(b) Residuals of query 5.

Figure 11: Evaluation of cardinality estimation on SP^{2}B queries.



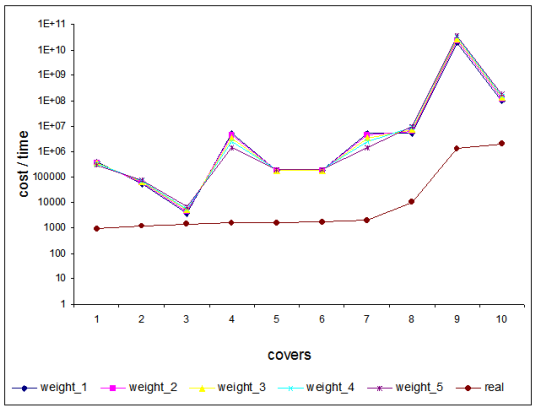
(a) Covers of query 6.



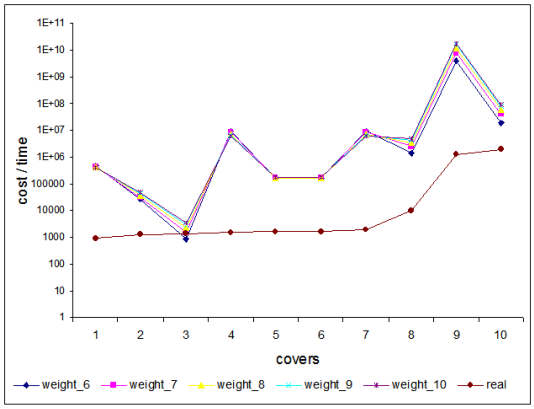
(b) Residuals of query 6.

Figure 12: Evaluation of cardinality estimation on SP^{2}B queries.

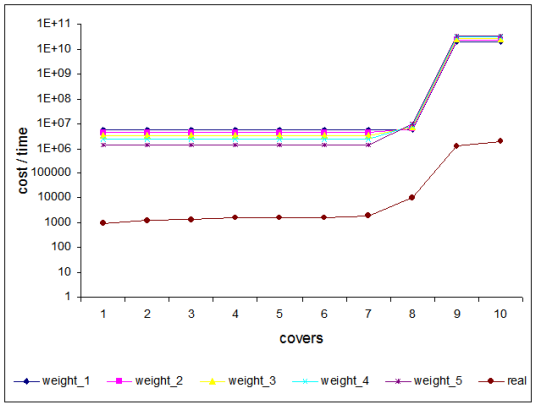
7 Evaluating Weighted Models



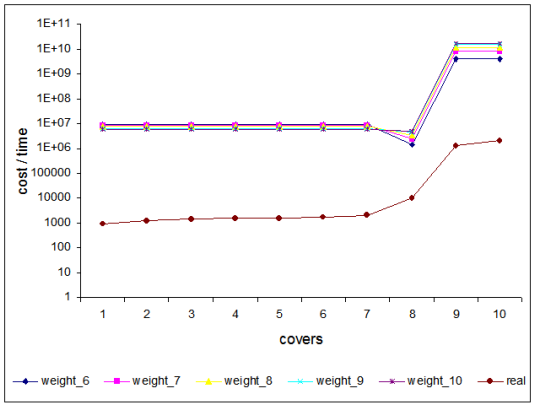
(a) Query 2 (BSBM)



(b) Query 2 (BSBM)

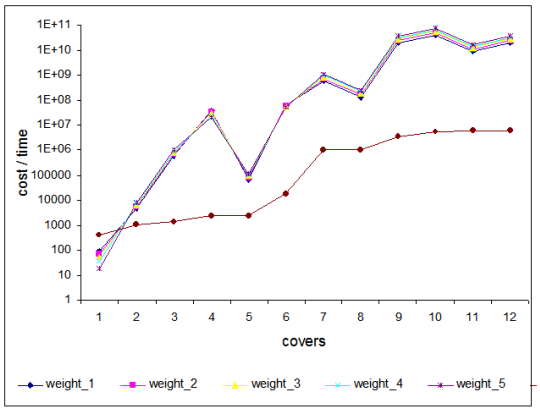


(c) Query 2 (BSBM)

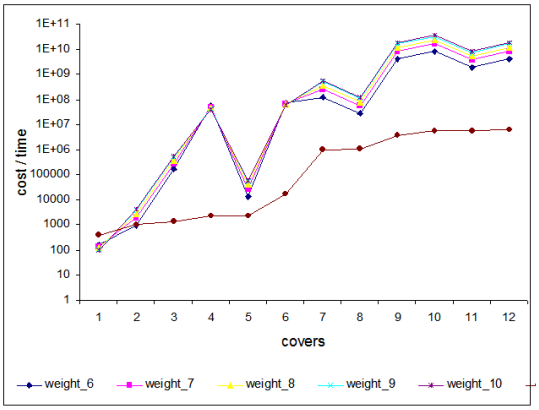


(d) Query 2 (BSBM)

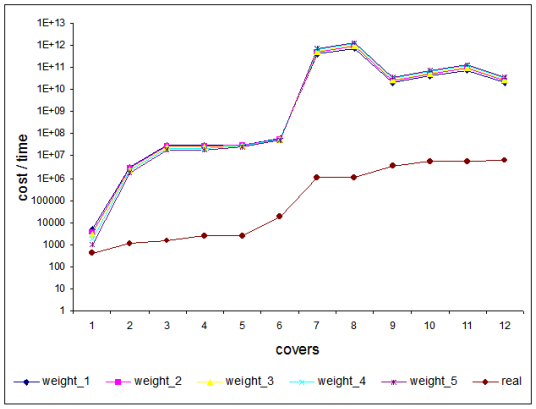
Figure 13: Comparison of estimated and real processing time for $query_2$ using SPOracle and $CardiOS_{mean}$.



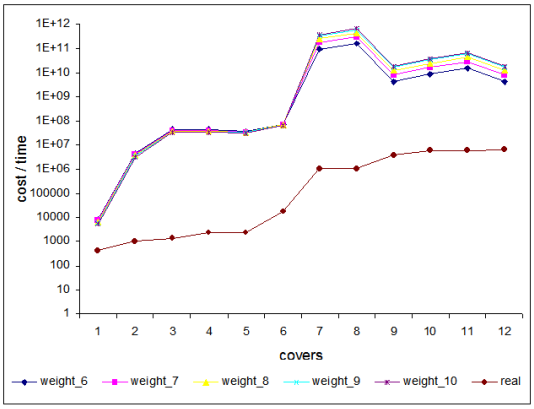
(a) Query 3 (BSBM)



(b) Query 3 (BSBM)

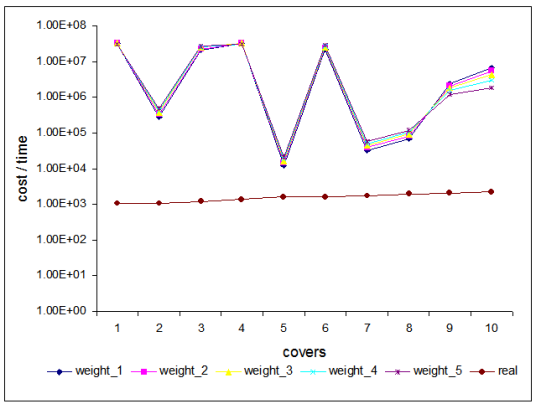


(c) Query 3 (BSBM)

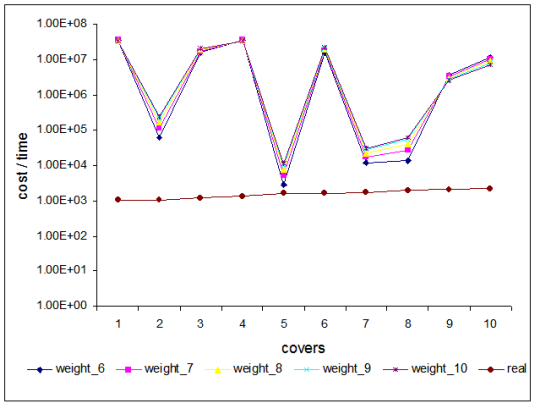


(d) Query 3 (BSBM)

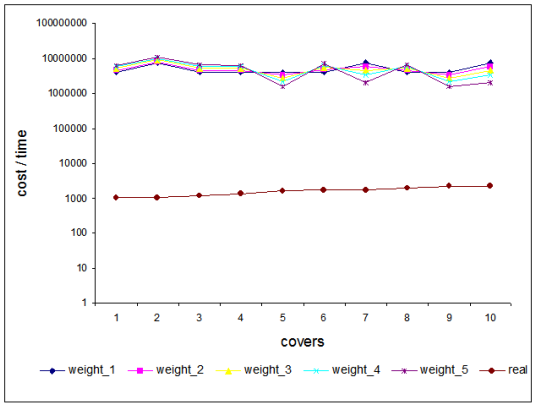
Figure 14: Comparison of estimated and real processing time for $query_3$ using SPOracle and $CardiOS_{mean}$.



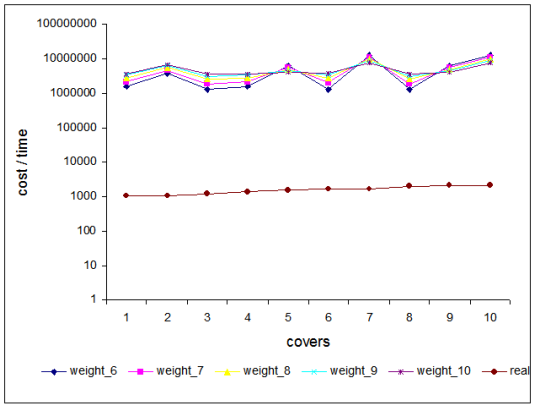
(a) Query 4 (SP2B)



(b) Query 4 (BSBM)

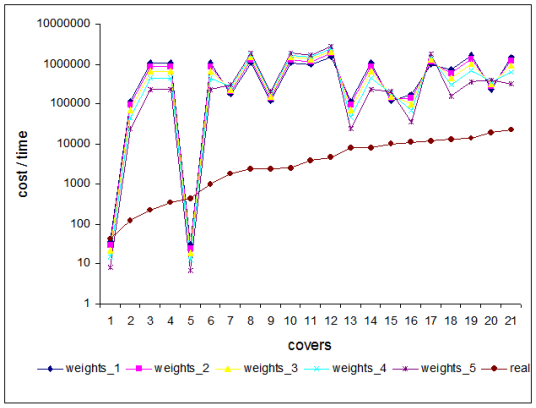


(c) Query 4 (SP2B)

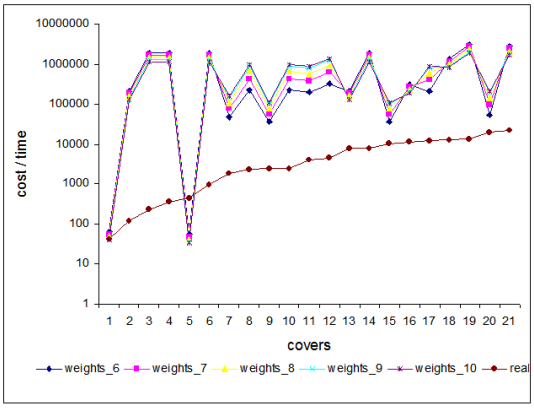


(d) Query 4 (SP2B)

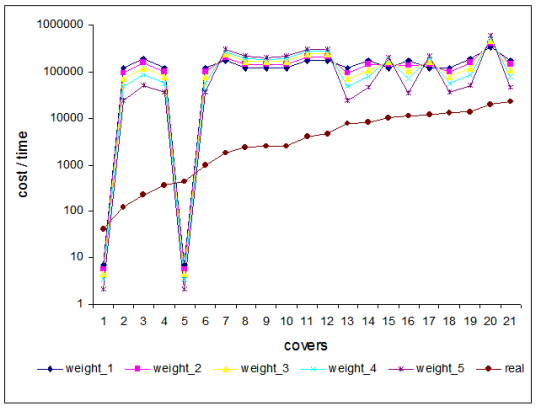
Figure 15: Comparison of estimated and real processing time for $query_4$ using SPOracle and CardiOS (mean). Estimations were done with 10 different weights over a 250k triples SP2B dataset.



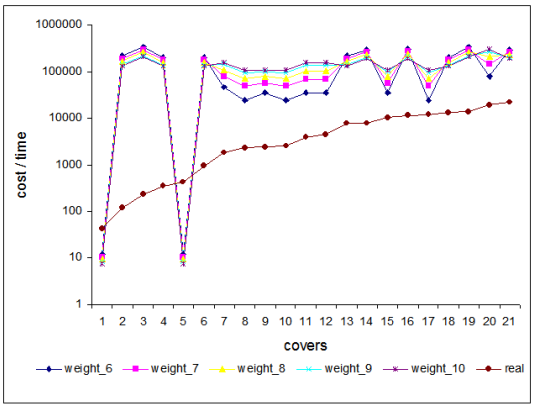
(a) Query 5 (SP2B)



(b) Query 5 (SP2B)

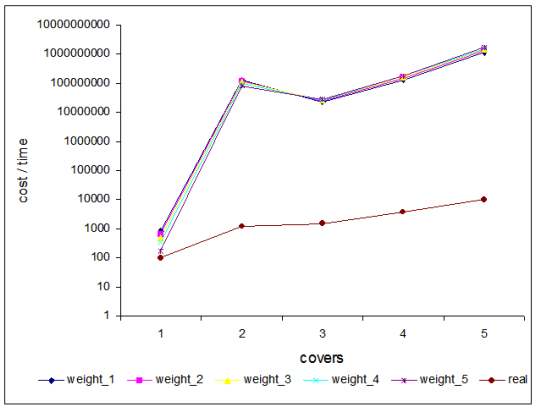


(c) Query 5 (SP2B)

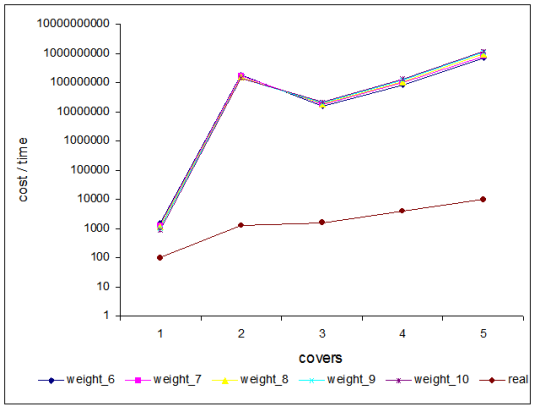


(d) Query 5 (SP2B)

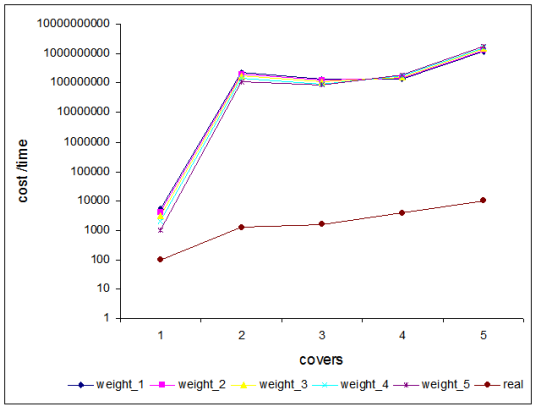
Figure 16: Comparison of estimated and real processing time for $query_5$ using SPOracle and CardiOS (mean).



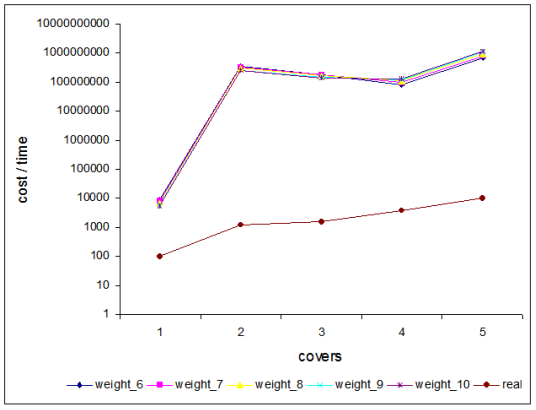
(a) Query 6 (SP2B)



(b) Query 6 (SP2B)



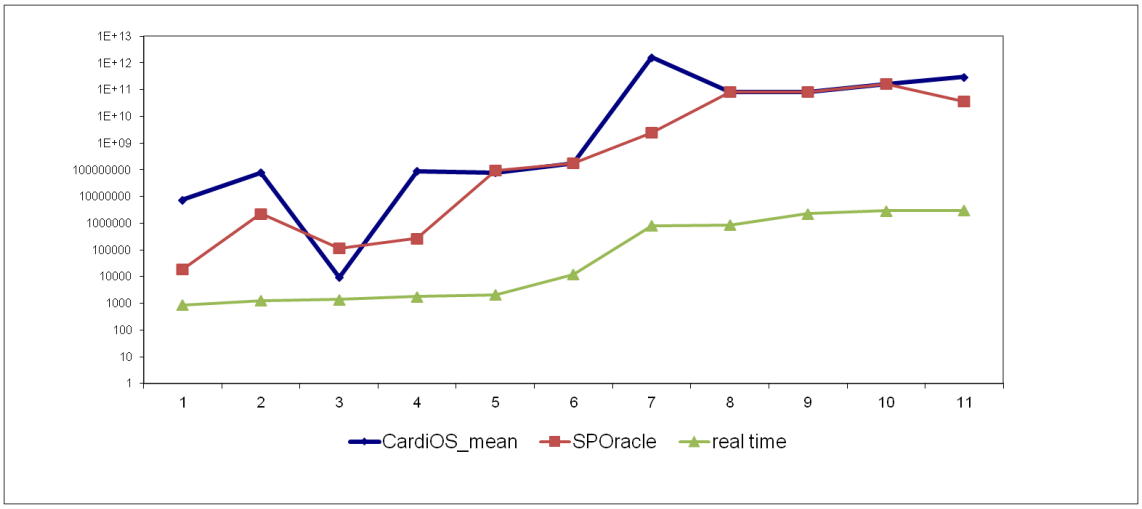
(c) Query 6 (SP2B)



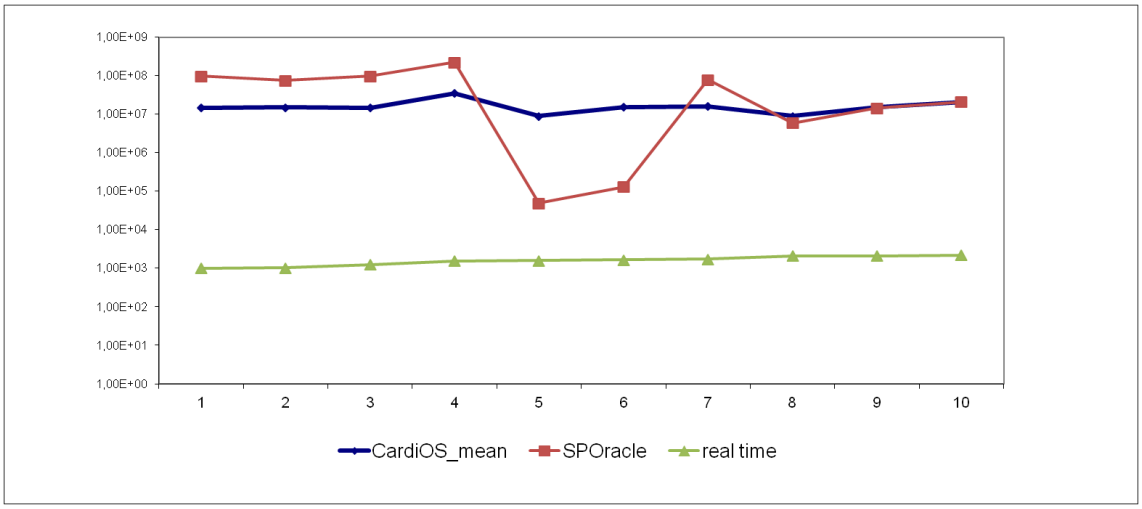
(d) Query 6 (SP2B)

Figure 17: Comparison of estimated and real processing time for $query_6$ using SPOracle and CardiOS (mean).

8 Accuracy of Cost Models

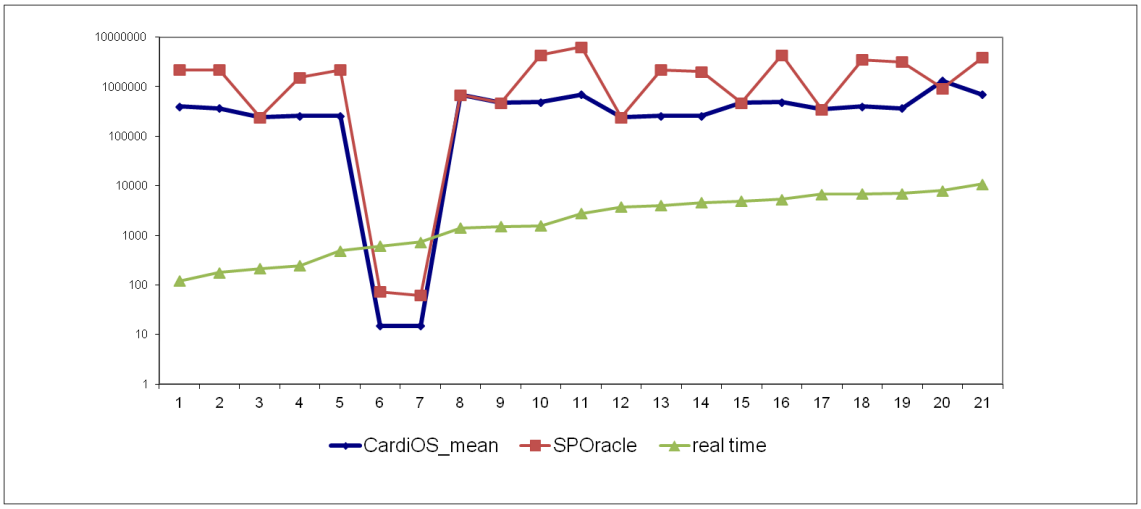


(a) Query 3 (BSBM)

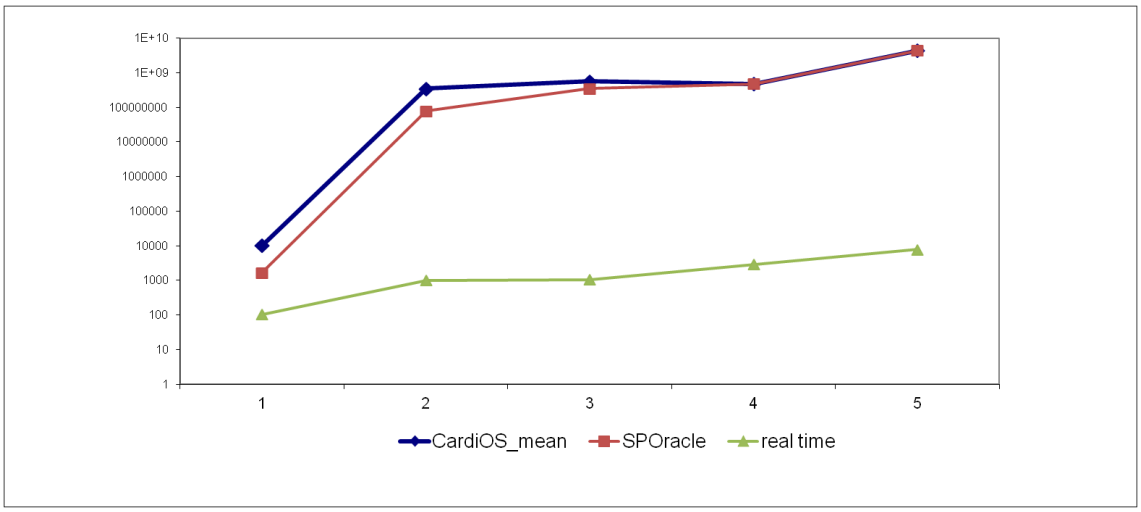


(b) Query 4 (SP2B)

Figure 18: Processing time vs estimates. For each test query all covers are estimated and executed using BSBM and SP2B.



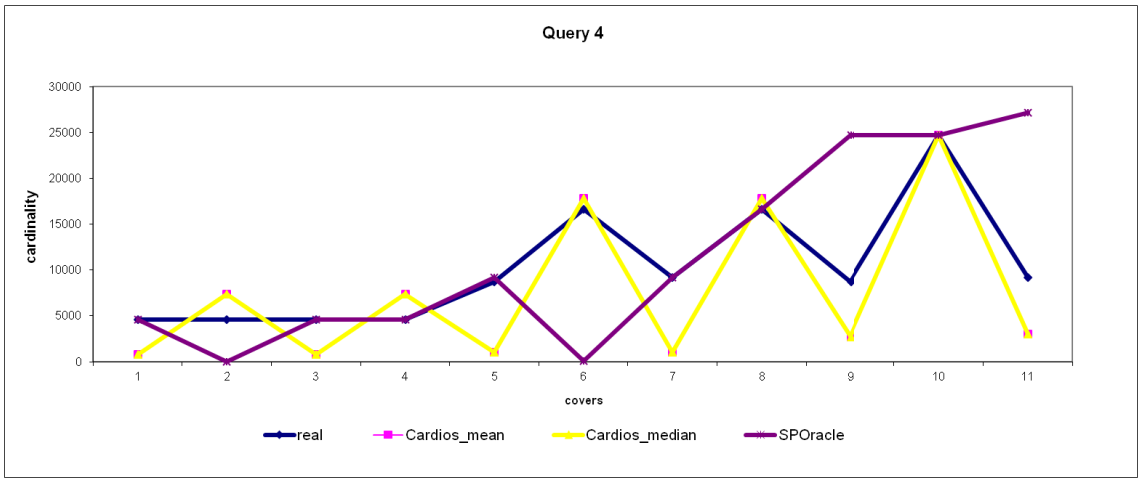
(a) Query 5 (SP2B)



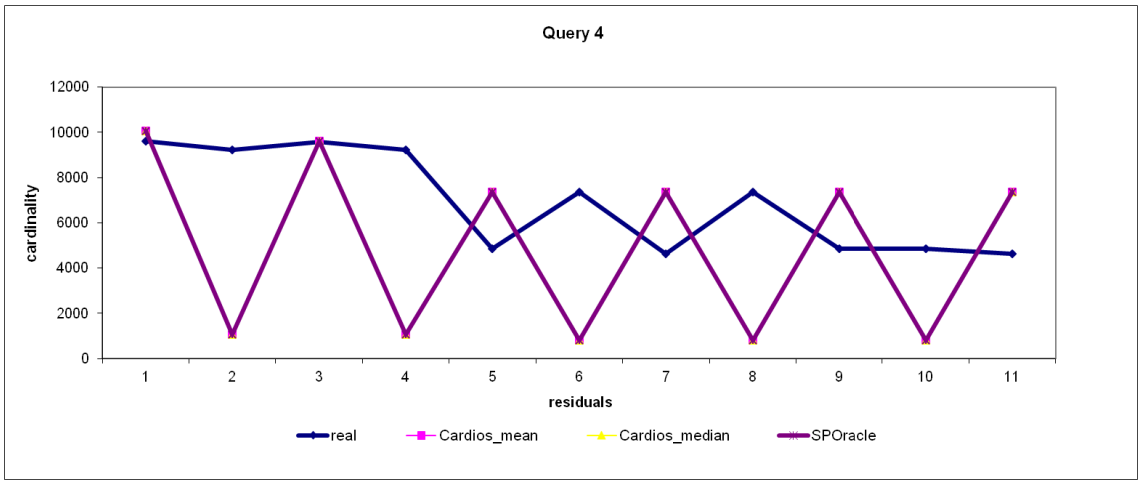
(b) Query 6 (SP2B)

Figure 19: Processing time vs estimates. For each test query all covers are estimated and executed using BSBM and SP2B.

List of Tables



(a) Covers of query 4.

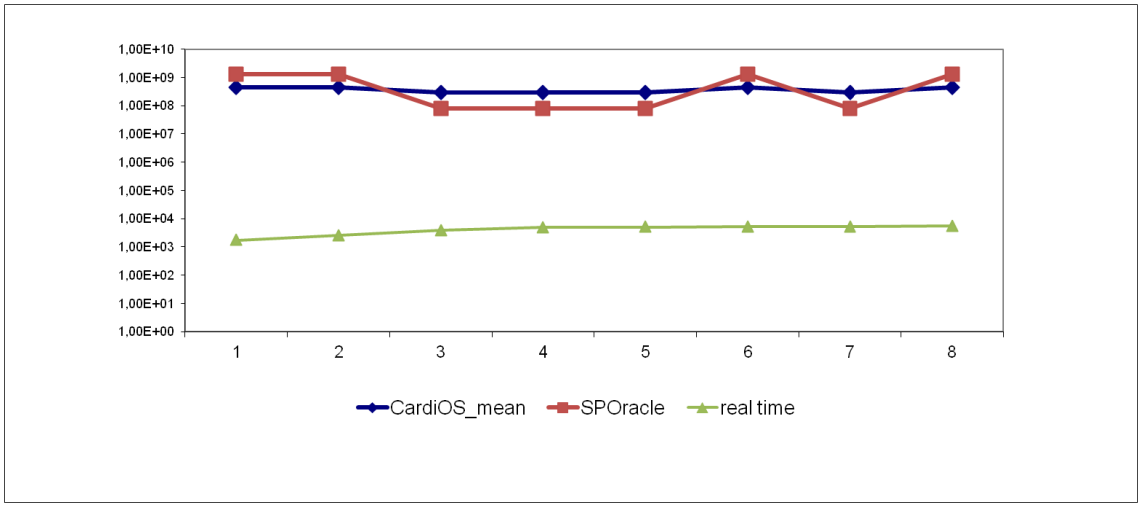


(b) Residuals of query 4.

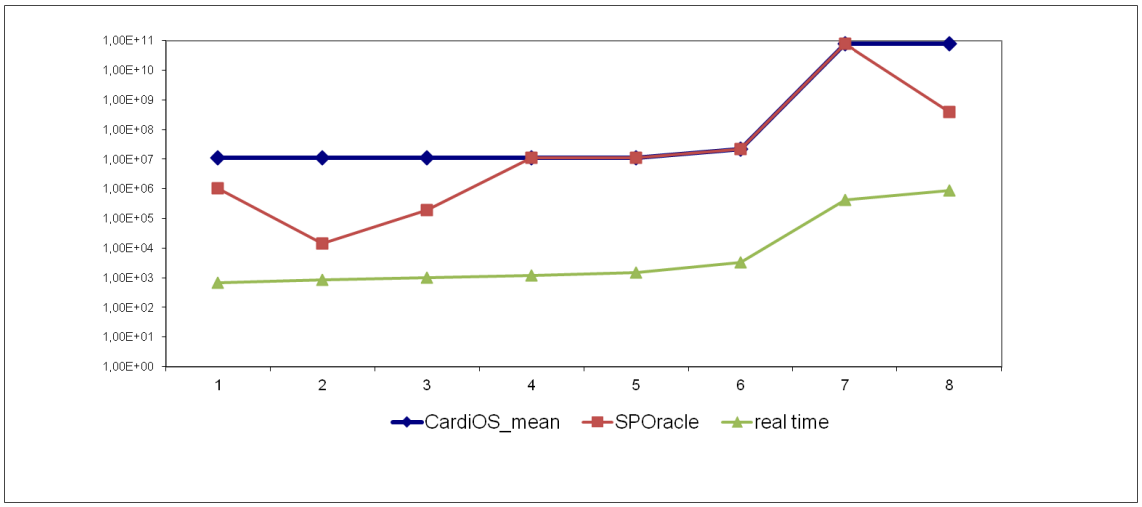
Figure 3.8: Evaluation of estimation on covers in query4 over SP^{2}B.

For instance, the *rdf:type* predicate contained in query1 has a minimal fan at subject of 1 and a maximal fan at subject of 4, the mean fan is 1.08 and the median fan is 1. The estimations based on these statistics remarkably differ from the real cardinality. Only very few subjects actually have four predicate edges with *rdf:type*, but all of them are product nodes. Unfortunately, this is exactly what query1 focuses on. Therefore, the average error of estimation of mean and median function of CardiOS is nearly 74%. On the other hand, SPOracle benefits from the cases where its cover estimates equal the real cardinality. Even when some cover estimates have errors, the mean error for the set of covers decreases noticeably.

as seen previously in Figure 3.6.



(a) Query 1 (BSBM)

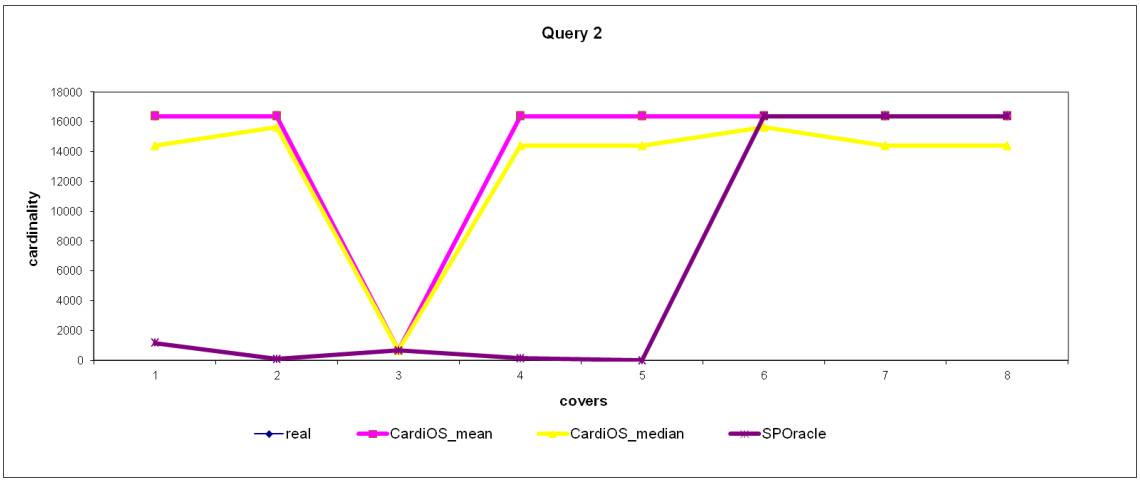


(b) Query 2 (BSBM)

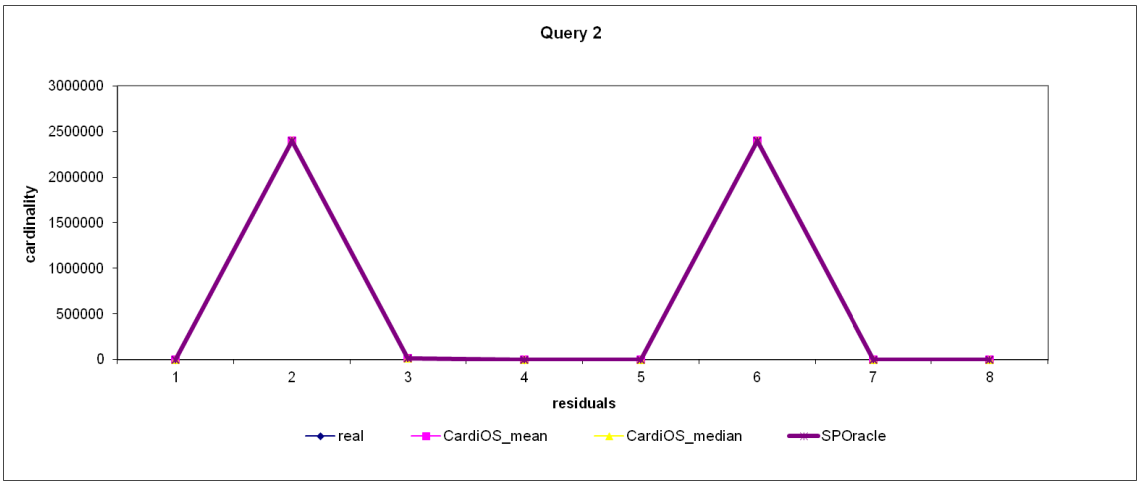
Figure 3.10: Processing time vs estimates. For each test query all covers are estimated and executed over a 250K triples dataset (BSBM)

Figure 3.10(b) shows that the weighted mean and median function estimate the same values for some covers (e.g. covers 1 to 5). These cases occur when the estimated cardinalities for the covered and the residual part are equal. Notice also that the real running times for these covers range from 692 to 1492 milliseconds although the cost model using both functions (mean and median) rated them as equal. This shows that the cost model does not take into account all conditions to calculate a time–proportional cost. Nevertheless, the figures also show that the cost model predicts with some degree of accuracy the

6 Cardinality Estimation: Covered and Residual Patterns

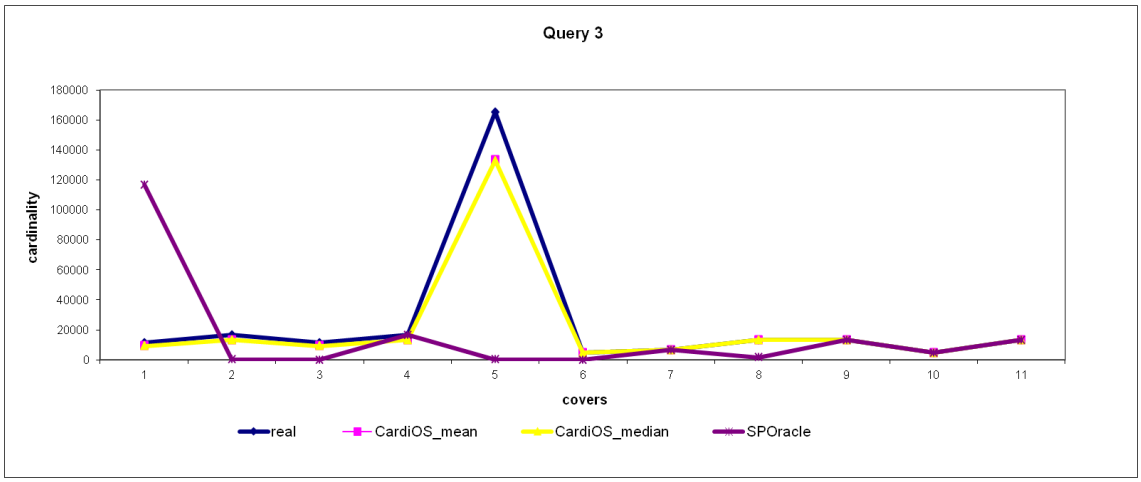


(a) Covers of query 2.

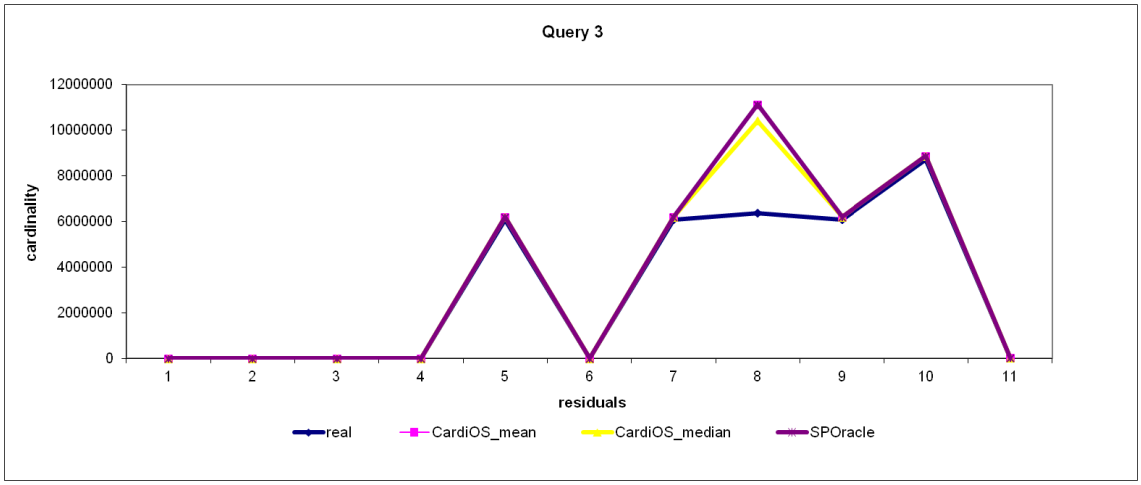


(b) Residuals of query 2.

Figure 9: Evaluation of cardinality estimation on BSBM queries.

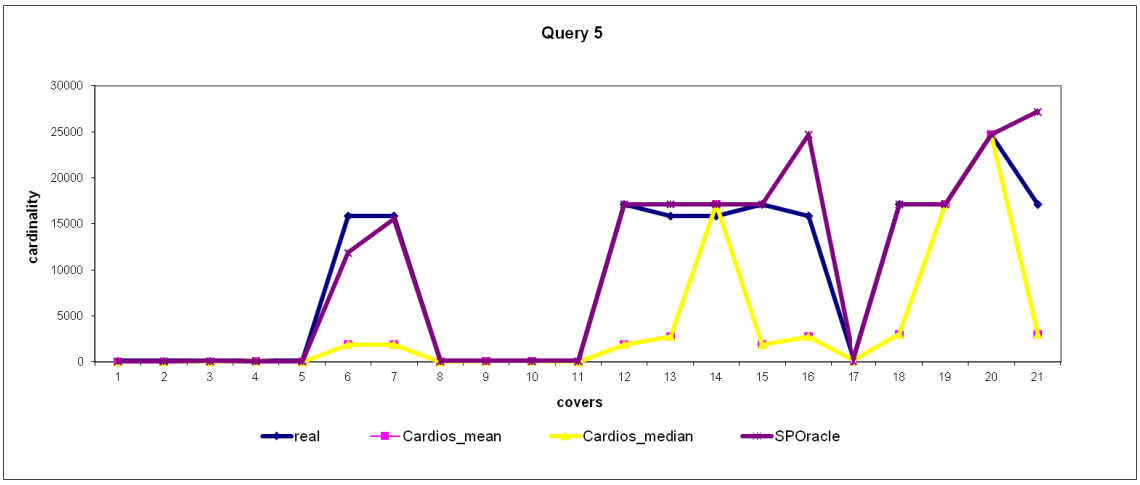


(a) Covers of query 3.

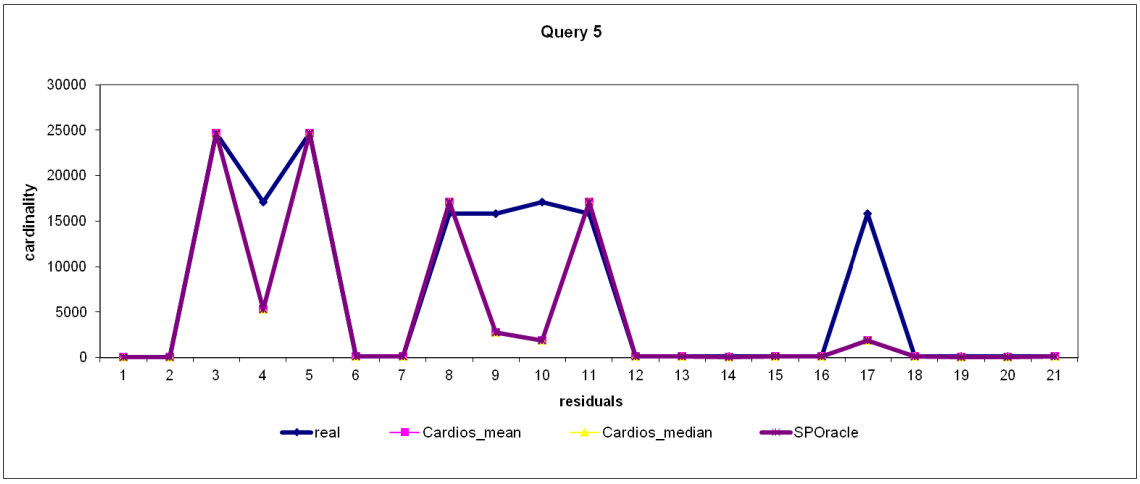


(b) Residuals of query 3.

Figure 10: Evaluation of cardinality estimation on BSBM queries.

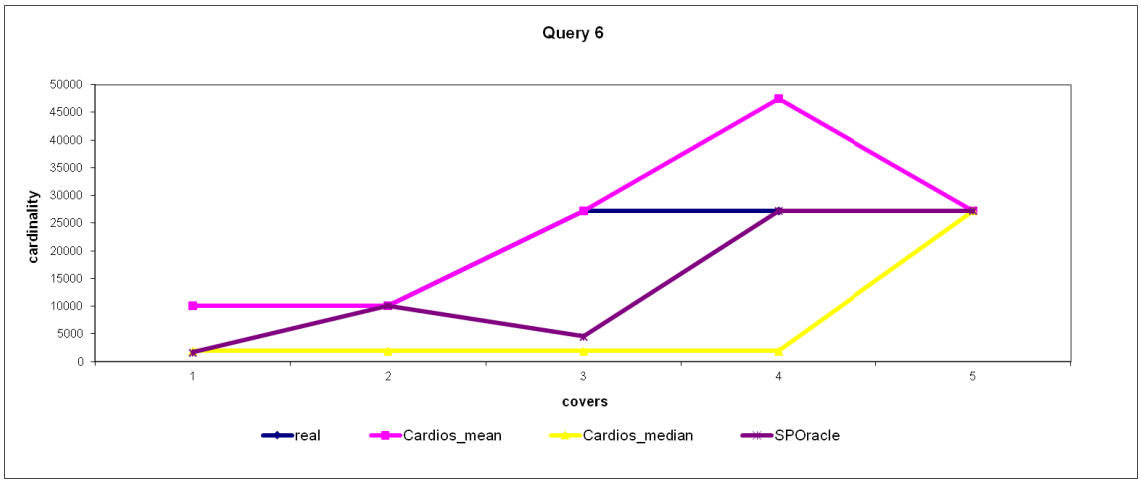


(a) Covers of query 5.

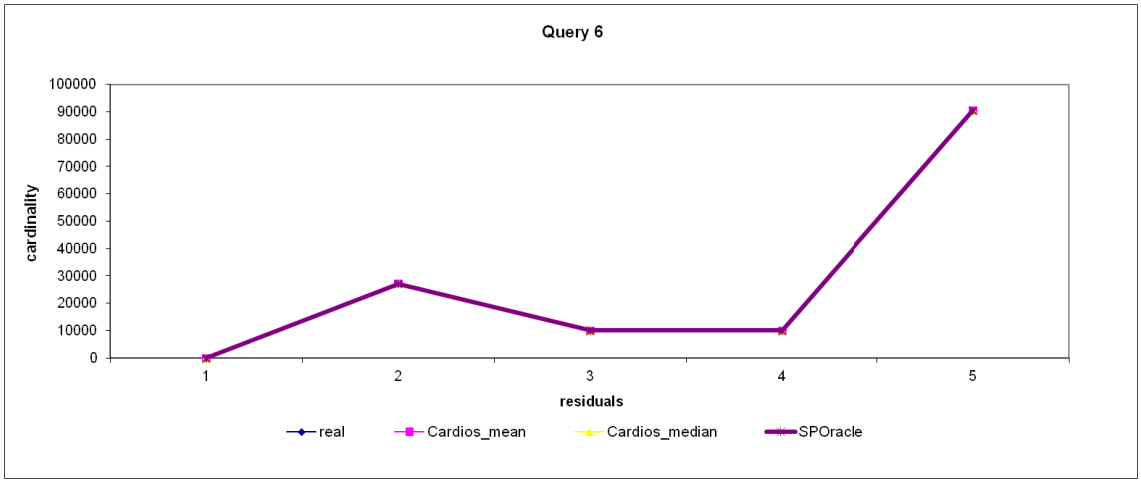


(b) Residuals of query 5.

Figure 11: Evaluation of cardinality estimation on SP^{2}B queries.



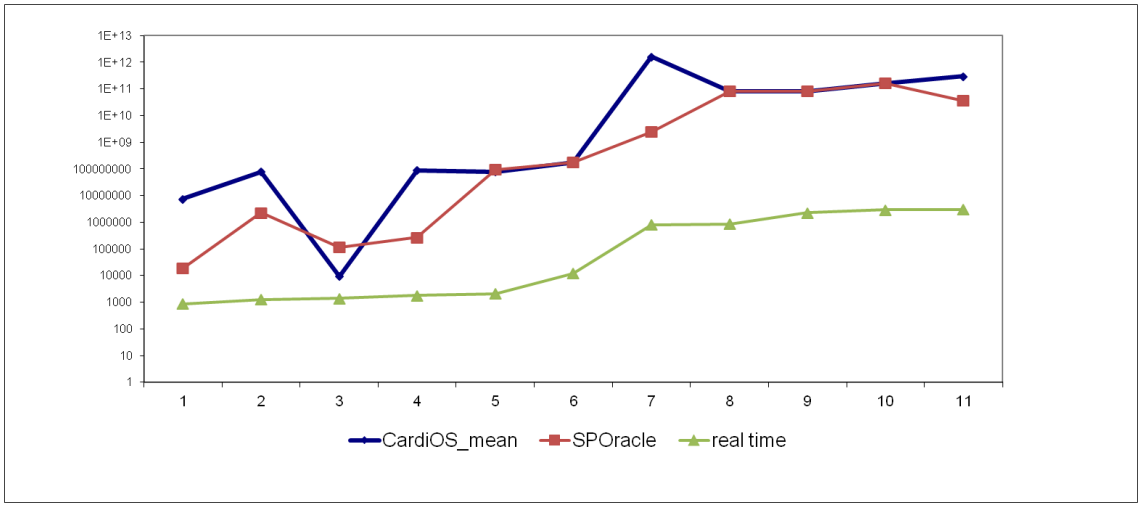
(a) Covers of query 6.



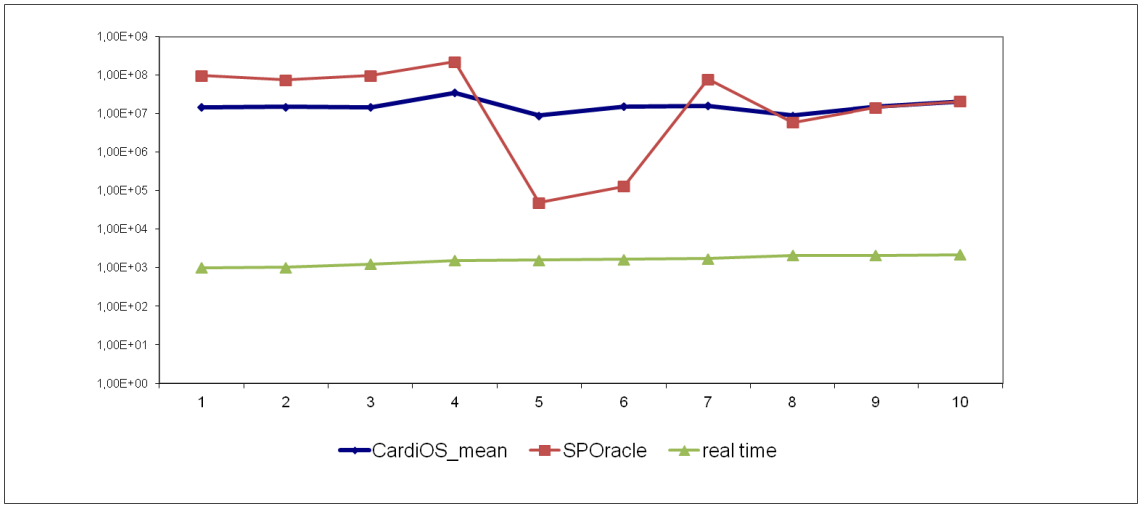
(b) Residuals of query 6.

Figure 12: Evaluation of cardinality estimation on SP^{2}B queries.

8 Accuracy of Cost Models

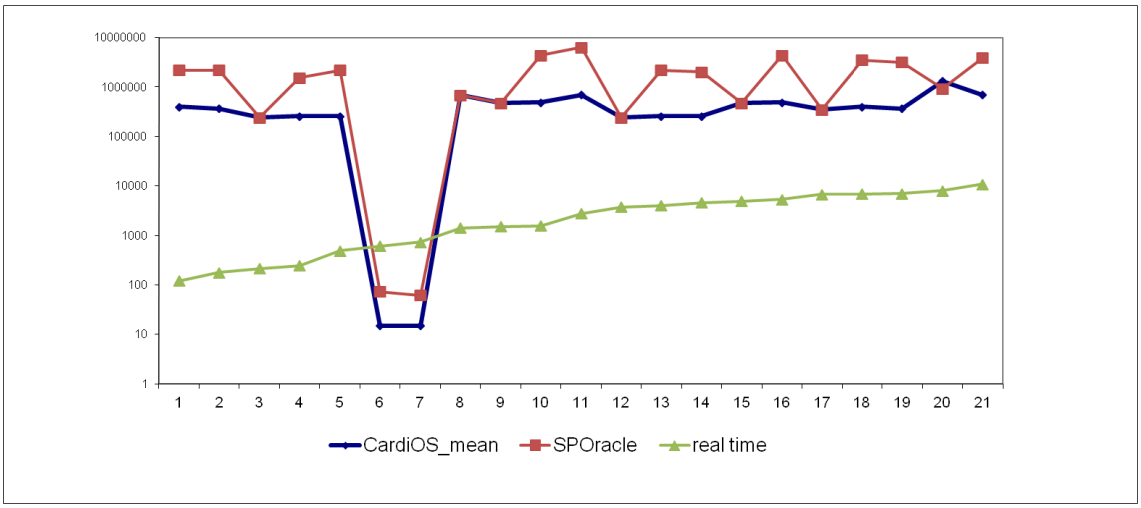


(a) Query 3 (BSBM)

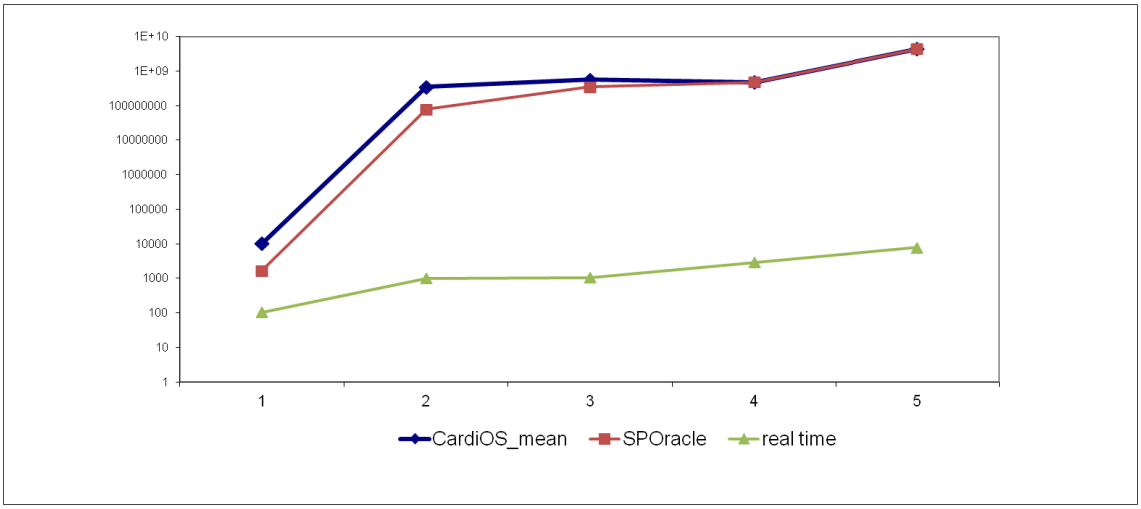


(b) Query 4 (SP2B)

Figure 18: Processing time vs estimates. For each test query all covers are estimated and executed using BSBM and SP2B.



(a) Query 5 (SP2B)



(b) Query 6 (SP2B)

Figure 19: Processing time vs estimates. For each test query all covers are estimated and executed using BSBM and SP2B.